Summer 8-28-2017

Integrated Environment and Proximity Sensing for UAV Applications

Shawn S. Brackett
University of Maine, shawn.brackett@maine.edu

INTEGRATED ENVIRONMENT AND PROXIMITY SENSING FOR UAV APPLICATIONS

By

Shawn Stevens Brackett

B.S. Engineering Physics, University of Maine, 2014

M.E. Engineering Physics University of Maine, 2017

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Engineering

(in Engineering Physics)

The Graduate School

The University of Maine

August 2017

Advisory Committee:

Sam Hess, Professor of Physics and Astronomy, Advisor

C.T. Hess, Professor of Physics and Astronomy

David Rubenstein, Research Associate Professor of Mechanical Engineering

Richard Eason, Associate Professor of Electrical and Computer Engineering

INTEGRATED ENVIRONMENT AND PROXIMITY SENSING FOR UAV APPLICATIONS

By Shawn Brackett

Thesis Advisor: Dr. Sam Hess

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Master of Engineering
(in Engineering Physics)

August 2017

As Unmanned Aerial Vehicle (UAV), or "drone" applications expand, new methods for sensing, navigating and avoiding obstacles need to be developed. The project applies an Extended Kalman Filter (EKF) to a simulated quadcopter vehicle though Matlab in order to estimate not only the vehicle state but the world state around the vehicle. The EKF integrates multiple sensor readings from range sensors, IMU sensors, and radiation sensors and combines this information to optimize state estimates. The result is an estimated world map to be used in vehicle navigation and obstacle avoidance.

The simulation handles the physics behind the vehicle flight. As a result of the motors there will be two primary forces acting on the vehicle; thrust and drag. These forces provide linear and rotational accelerations. These values are integrated to determine vehicle flight path and the vehicle state. Coupled into the integration process is the ability to control motor speed. Thus, a controller is built into the vehicle such that desired vehicle states (position, velocity, and orientation) can be achieved. Additionally, sensing methods are simulated following known detection procedures and statistics to provide realistic sensing models.

The EKF is used here due to non-linear components of the estimation model. The estimation model includes the states for the vehicle, nearby world objects, and nearby radiation measurements. By

combining IMU and GPS data with additional measurement data for an EKF, an estimation method attempts to combine all three types of information to build state estimates with minimized error. The EKF outputs the estimated state including vehicle, obstacle, and radioactive source locations. This information is used to avoid collisions and to identify and localize radioactive sources.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Sam Hess for allowing me to be his graduate student and conduct this research under him. I would like to thank the committee members, Dr. C.T. Hess, Dr. David Rubenstein, and Dr. Rick Eason for taking the time to assist me and not tearing me apart during the committee meetings. I'd like to thank my wife Irene Brackett for supporting me over the final months of the thesis completion. I would like to thank my parents Richard and Linda Brackett for assisting me over the many years. I would like to thank Eley and Caroline Wagnon for all their support down in GA. I would like to thank the rest of the Physics Department, the faculty and staff for their support, especially Pat Byard who has been essential part of my time at the University of Maine. I would also like to thank my peers and friends, Andrew Nelson, Desiree Nelson, Alex Khammang, Bill Cooley, Matt Valles, Michael Butler, Matt Parent, James Deaton, Stuart Lawson, David Stewart, Scott Mitchel, Jahrel Registe, Alex Moser, Zach Sawyer, Jessica Hamilton, William Smith, Josh Fuhrer, Brad Boden, Spencer Mishoe, Jamie Hutchins, Megan Helm, Kimberly Picard, Simone Dubay, Mike Heikkinen, Chris Camire, and Henry Moniz.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Drones, or unmanned aerial vehicles (UAVs), have been on the minds of humans for hundreds of years. While early drone applications may be unrecognizable as such to a modern drone enthusiast, one cannot deny their existence. Drones have particularly been developed early on for military applications. Early applications for combat drones were as simple as strapping bombs onto a swarm of balloons, and later on applications evolved to surveillance as well as destructive payload delivery. A little more than 5 years ago saying the word "drone" would typically conjure up images of an unmanned military vehicle such as a Predator drone. Over the years as technology developed so too did the "drone". Today with radio communication one can send and receive advanced commands and data to and from these remote vehicles. As drone technology was brought to domestic applications such as border patrol and policing, so too was it brought to the consumer. Drone kits, similar to the Matrix-S drone discussed briefly, can be bought and assembled by anyone. Open-source software including ground stations and autopilots exist and are freely accessible to anyone willing to put in the time and effort necessary to learn it. Additionally, packages are available where the drone is ready to fly practically out of the box (some sticker removal required). While a fixed wing drone design resembles a traditional airplane with an airfoil used to generate the vehicles lift, the most popular drones at the moment are multirotor aircraft. A multirotor drone design has more similarity to a helicopter, and uses multiple propellers to lift the vehicle up, adjusting motor rates results in moving the vehicle in various ways.

Drones, particularly multirotor drones are highly maneuverable compared to a fixed wing aircraft. Their motion can be easily controlled in terms of speed and direction; unlike a fixed wing vehicle they are not limited to forward motion. It is because of their maneuverability that they have gained a great deal of interest over the years. Today multirotor vehicles are often used for photography, cinematography, racing, and hobbyist flying. However, their applications are rapidly expanding. Take for example Amazon, which is working to develop an unmanned drone package delivery system, and in

fact has already begun testing. In March of 2011 a massive earthquake caused a Tsunami to slam into the Fukishima Daiichi Nuclear Reactor, which caused major radiation releases into the surrounding area. At that time, drone applications were still quite new, but multirotor drones were used to fly around the area to search for radioactive leak sites. These two applications have something in common: they are navigating drones in a lower altitude environment, potentially in proximity to obstacles and/or people. Autopilots are now an integral part of the UAV, but they typically function to control vehicle stability, altitude, speed, and direction of motion, often without consideration of the location of obstacles. At higher altitude, the risk of obstacles is relatively low (except for manned aircraft). As the vehicles come closer to the ground, suddenly obstacles begin to cause serious issues with navigation.

Taking inspiration from operating in a low altitude hazardous (to humans) environment, this work aims to develop an algorithm that integrates a variety of on board sensor data to estimate the vehicle's position, to find the location of objects within the surrounding environment, and to estimate the location of radioactive sources within the environment. Three separate methods were developed. The core of the first algorithm uses a Kalman Filter (KF) and other routines which process raw measurements into a form more compatible with the KF's world design. The second algorithm was an attempt to improve the first, which instead makes use of an Extended Kalman Filter (EKF). The second algorithm also uses similar routines for processing raw measurements into better format for the EKF design. The third algorithm which underwent the most radical design change again uses an Extended Kalman Filter. It has some different routines which also manage raw data so it is once again compatible with the EKF's design components.

In order to develop and test each algorithm, a simulation was built in Matlab to model vehicle flight, vehicle autopilot and controls, proximity sensing, and environment sensing. The world is modeled by cubes representing a portion of volume in the world, each cube has a value corresponding to whether it is occupied or not. The proximity detection routines query these cubes as it simulates a

range finding beam relative to the vehicle. The radioactive sources are simulated by placing one or more point sources in the environment. The activity and subsequent counts recorded by a Geiger counter in the environment follow Poisson statistics. For the vehicle flight and controls, an LQR controller is used to control the four thrust vectors associated with the quadcopter model; in addition, it also controls the yaw rate of the vehicle. These controls directly contribute to the vehicle's equations of motion which are numerically integrated to compute the vehicle path. The algorithms developed are tested over the simulation and compared to actual vehicle flight path and actual environment status.

As the algorithm designs develop, the vehicle state, environment estimates, and radiation estimates are somewhat tied together. This is done in hopes that the different measurements and estimates would work in unison to improve upon each other. Due to all measurements being made by the vehicle, all measurements are connected to the vehicle states. The effect is a double edged sword as it can have a positive and negative effect. Most of the algorithms designed successfully generate fairly accurate vehicle estimates, a useable world map, and work towards localizing the radioactive source. In the case of the 1st and 2nd algorithms, they estimate radiation measurements, which are used to estimate potential source location. The 3rd algorithm attempts to estimate both source location and radioactivity level.

Chapter 2 will begin by building the key elements of the simulation. The methods section starts by developing the equations of motion to be integrated by the 4th order Runge-Kutta ODE solver. From the equations of motion, the linear and rotational accelerations of the vehicle can be determined. A 3-2-1 Euler Rotation is built for a rotating vector from the inertial frame to the body frame, and the inverse matrix can be used to go from the body frame to the inertial frame. With the integration of the vehicle motion, the controls are also applied to steer the vehicle to different waypoints. Two layers of controls are implemented, the first is an LQR controller which provides a gain matrix applied to error signals which modify the vehicle thrust vectors and turning. The second layer allows for slightly more

control over the error signals to be sent to the LQR controller. This second layer is important for making use of maneuver profiles and smoothing out the vehicle flight.

The next key components discussed in the methods second are how the simulation of the environment and the simulation of how the vehicle sensors detect the environment. There are two aspects to the world, physical obstacles and radioactive point sources. The physical world is represented by small cubes with values corresponding to occupancy. The radioactive point sources are represented by a location and activity in counts per second. When detecting the physical environment a laser range scanner similar to range data from an X-Box Kinect or LiDAR sensor providing range data in a frontal cone in the vehicle's forward direction. To check for occupancy, the "beams" are used to reference the cube occupancy and determine in an obstacle is detected. In order to describe radiation detection, a simplified Geiger counter model is used. This model has no directional dependence of the detection, and follows Poisson statistics for the counts per second based on the location(s) and activity of each of the radioactive source(s). The simulation also assumes a number of other sensors attached to the vehicle that the algorithms will have access to; these include: GPS, IMU, Gyro, and compass devices.

With the environment and vehicle movement handled, the next component is the process which estimates the vehicle and the environment. The Kalman Filter process is introduced and described some some detail. The Kalman Filter and the Extended Kalman Filter is used to estimate the vehicle position and the environment. The conceptual and mathematical flow is described for the (Extended) Kalman Filter. Once the KF concept is introduced the three different applications are described - KF 1, EKF 2, and EKF 3. Each application has unique components of their design as they address varying issues seen during their progression. Those issues include computation time, the size of the volume estimated, accuracy, and overall feasibility. The transition and measurement models are described along with additional important information regarding their application.

The results section presents general results for the first two applications. The first KF 1 application shows strong capabilities in environmental estimation and obstacle avoidance, however issue arise with the size of the volume estimated, computation time, and radiation estimates do not provide a great deal of information regarding source location. The second EKF 2 which addresses computation time and estimation volume provides very poor results with world estimation and accuracy. The final EKF 3 model is presented as the final algorithm developed address most issues and shows the most overall promise. It has successful obstacle avoidance up until GPS error of 3m, and is capable of localizing a radioactive source to within less than 3m of error. Additional tests and results are presented to look at the feasibility of the EKF 3.

In the discussion, the work presented is compared with some other relevant work. In addition, the results presented are discussed in greater detail. The strengths and weaknesses of each application are discussed. Various important issues with the work are talked about with potential solutions. Potential for improvement of the algorithms is discussed, and ways of applying the work to a real world system are also described. The results are a very strong algorithm which handles environmental and vehicle estimation. The algorithm is able to be used to influence the vehicle position and velocity controls for obstacle avoidance and navigation towards potentially identified sources.

CHAPTER 2: METHODS

Experimental data was simulated with code written in Matlab using both built-in Matlab functions and custom functions. This section breaks down the different parts of the simulation code and conceptualizes it. The first key concepts discussed are the equations of motion for the vehicle and how they are integrated in the simulation. Embedded into the integration process is a portion of the vehicle controls. These controls are also explained. With the equations of motion and the vehicle controls described, a Runge-Kutta ODE solver is utilized for integration. As a result, a method of calculating and controlling the vehicle's actual flight path is formed.

Next, the creation and detection process for the real world must be discussed - how the world information is represented and how that world's spatial and radioactive information is acquired through simulated sensing methods. For the spatial detection a 3-D range sensor is modeled, and for the radioactive data, a simple Geiger counter model is used. This section will discuss how the code mimics both types of sensors, from measuring object distance to integrating counts from radioactive sources using Poisson statistics.

Next is the estimation process itself. To estimate the vehicle and world state, an Extended Kalman filter is used. First the simple theory behind the Kalman and Extended Kalman filter is introduced, followed by the specific application in this simulation. The application talks about the specific state vector used, the estimation model, and the measurement model. By taking the estimated world obstacles and computing the gradient, a newer "safer" trajectory can be computed and passed to the vehicle control algorithm.

2.1 EQUATIONS OF MOTION

The most fundamental part of the program is the simulation of drone flight. To describe the Equations of Motion the inertial x_i, y_i, z_i frame and the Body x_b, y_b, z_b frame must be explained. Figures 2.1 and 2.2 illustrate the body frame and its relation to the inertial frame. The

vehicle's forward facing x_b-direction is the axis about which the vehicle roll rate is referenced. The pitch rate axis corresponds to the y_b-direction. Vehicle yaw rate is about the z_b-axis. It is more useful to use the vehicle heading, which in the context of this work will be defined as the direction the vehicle x_b-axis points relative to the inertial frame's x_i-axis.

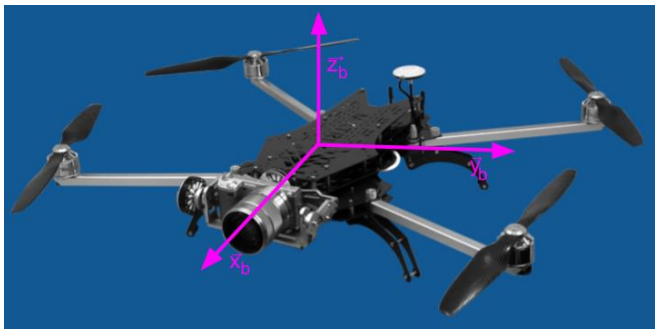

Figure 2.1 Matrix S Drone w/ Body Frame: Image of Turbo Ace Matrix S Drone with Body frame coordinate system.

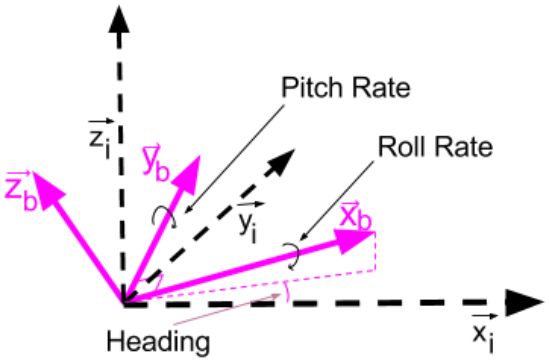

Figure 2.2 Body Frame and Inertial Frame: Body frame coordinate system as seen in an Inertial coordinate system with roll, pitch, yaw.

For the multi-rotor quadcopter model that was used, there are two forces that will dominate the motion: lift and drag. For simplicity only the lift force was simulated. For a quadcopter model the lift force dominates five of the six degrees of freedom. It does not influence the vehicle yaw. Vehicle yaw

handling will be discussed later. Four separate force vectors representing the thrust forces from each rotor are summed up in the vehicle frame. Due to these forces not being applied towards or at the vehicle's center of mass a net torque must also be considered, hence influencing vehicle roll and pitch. Equations 2.1 and 2.2 shows the Net Lift Force and Net Torque equations as referenced in the vehicle body frame.

$$\vec{F}_{Lnet_{body}} = \sum_i \vec{F}_{Li} \quad \text{Eq. 2.1}$$

$$\vec{T}net_{body} = I\vec{\alpha} = \sum_i \vec{r}_i \times \vec{F}_{Li} \quad \text{Eq. 2.2}$$

The net torque can be divided by the vehicle's moment of inertia to acquire the accelerations of the vehicle's roll, pitch, and yaw as shown in equations 2.3 and 2.4. However, these are not the Euler Angle rates. The Euler angles, with the exception of the first rotation (psi), are referenced in interim coordinate systems. The body frame accelerations are not the Euler accelerations.

$$\vec{\alpha}_{net_{body}} = \frac{\vec{T}net_{body}}{\mathrm{I}} = \begin{bmatrix} r\ddot{o}ll \\ p\ddot{\imath}tch \\ y\ddot{a}w \end{bmatrix} \neq \begin{bmatrix} \ddot{\varphi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} \quad \text{Eq. 2.3}$$

The body frame accelerations can be integrated to acquire the body frame rates, p, q, and r. These rates can be transformed to Euler rates which can be integrated for Euler angles to be used in the DCM for rotating between the inertial and body frames. Equation 2.4 shows the transformation from the orthogonal frame rate vector to the non-orthogonal Euler rates vector.

$$\begin{bmatrix} \dot{\varphi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = L' \begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & \sin(\varphi)\tan(\theta) & \cos(\varphi)\tan(\theta) \\ 0 & \cos(\varphi) & -\sin(\varphi) \\ 0 & \sin(\varphi)\sec(\theta) & \cos(\varphi)\sec(\theta) \end{bmatrix} \begin{bmatrix} r\dot{o}ll \\ p\dot{\imath}tch \\ y\dot{a}w \end{bmatrix} \quad \text{Eq. 2.4}$$

In the simulation, the assumption that the Euler angles are about equal to the roll, pitch, and heading angles. This is an incorrect assumption to make; however, with the heading (psi) referenced in the inertial frame, and very small roll and pitch maneuvers lasting for short periods of time, results with and without the assumption perform relatively the same.

Referencing figure 2.3, it is important to understand that with upward lift forces in the vehicle frame, and with no W-axis component on the "r" vectors to the lift forces from the center of gravity there is no torque about the vehicle's W-axis hence yaw double dot will always equal zero.

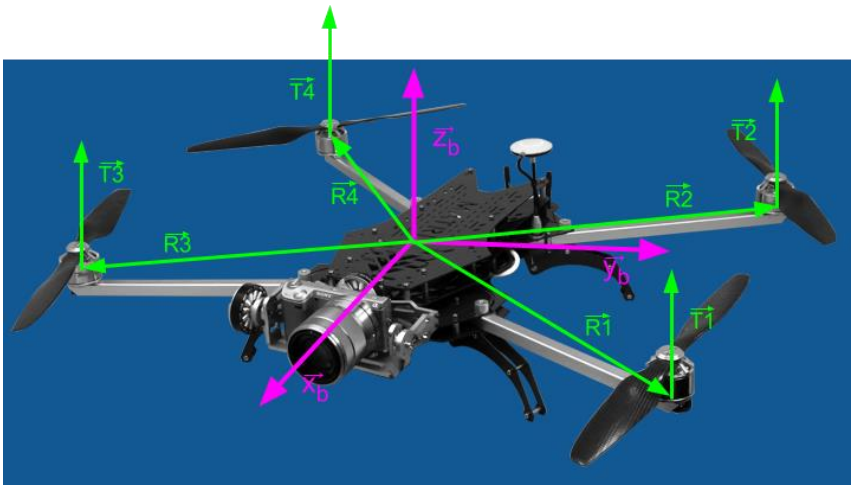

Figure 2.3 Matrix S w/ Thrust Vectors: Image of Turbo Ace Matrix S drone with Body frame coordinate system, thrust forces and R vectors for torques.

The translational accelerations are integrated in an inertial x_i, y_i, z_i frame. The net lift force vector must first be rotated into the inertial frame using a DCM computed from the vehicle's roll, pitch, and heading Euler angles. In order to rotate from the inertial x_i, y_i, z_i frame a 3-2-1 Euler rotation is applied. Equations 2.6, 2.7, 2.8, 2.9 illustrate the rotation process given the rotation matrix from Earth to Body Frame R shown in equation 2.5

$$ {}_{B}R_{E} = R_1 R_2 R_3 \qquad \text{Eq. 2.5} $$

$$ R_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi \\ 0 & -\sin\varphi & \cos\varphi \end{bmatrix} \qquad \text{Eq. 2.6} $$

$$ R_2 = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ sin\theta & 0 & \cos\theta \end{bmatrix} \qquad \text{Eq. 2.7} $$

$$R_3 = \begin{bmatrix} \cos\psi & sin\psi & 0 \\ -sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{Eq. 2.8}$$

$$\vec{F}_{Lnet_{earth}} = {}_BR_E^T \vec{F}_{Lnet_{body}} \qquad \text{Eq 2.9}$$

To get the net Force on the vehicle the rotated net left Force is added to the force of gravity. After dividing by the vehicle mass the linear accelerations can then be integrated in the inertial x_i, y_i, $z_i$frame. Equations 2.10, 2.11, and 2.12 illustrate this process and give us the final equations of motion for five degrees of freedom.

$$\vec{F}_{net_{earth}} = m\vec{a} = \vec{F}_{Lnet_{earth}} + \vec{F}_g \qquad \text{Eq. 2.10}$$

$$\vec{a}_{net_{earth}} = \frac{\vec{F}_{Lnet_{earth}}}{m} + \vec{g} \qquad \text{Eq. 2.11}$$

$$\vec{a} = \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} \qquad \text{Eq. 2.12}$$

2.2 THE INTEGRATION PROCESS/COMMANDS AND CONTROLS

The Controls and Integration Process are intertwined together. The process is repeated many times throughout the simulation. It plays the key role of determining where the vehicle is, and how it will move. We will go in greater detail below but the general flow is as follows. First the Runge-Kutta solver function is called (rk4). The function that defines the derivatives is the culmination of the vehicle's equations of motion (EOM2). The thrust vectors are determined through a closed-loop system that attempts to obtain a specified vehicle state – this vehicle state is defined outside the rk4() function. The EOM2() function calls the controls() function which determines a gain matrix. With the gain matrix the EOM2() function determines the change in the vehicle's control vector from trim state. With the vehicle state and control vector determined, the rates are computed and integrated.

The Integration Process uses the above equations of motion and integrates them. It uses a 4th order Runge-Kutta method to numerically integrate the equations of motion. The code for the 4th order Runge-Kutta method was primarily obtained from (Sauer 2012). The appendix code (APPENDIX B) which

contains the slightly modified version of the Runge-Kutta used here. The Runge-Kutta is a numerical method of solving ordinary differential equations, much like those built by the vehicle's equations of motion. In general, the 4th order Runge-Kutta works well given the function and the initial value:

$$\dot{x} = f(t, x) \qquad \text{Eq. 2.13}$$

$$x(t = 0) = x_0 \qquad \text{Eq. 2.14}$$

The goal of the 4th order Runge-Kutta is to provide a solution for x some timestep "h" later. Its solution is the following equation:

$$x(t + h) = x(t) + \frac{h}{6}(s_1 + 2s_2 + 2s_3 + s_4) \qquad \text{Eq. 2.15}$$

S1, s2, s3, and s4 are computed mathematically and in order as follows:

$$s_1 = f(t, x) \qquad \text{Eq. 2.16}$$

$$s_2 = f(t + \frac{h}{2}, x + \frac{h}{2} * s_1) \qquad \text{Eq. 2.17}$$

$$s_3 = f(t + \frac{h}{2}, x + \frac{h}{2} * s_2) \qquad \text{Eq. 2.18}$$

$$s_4 = f(t + h, x + h * s_3) \qquad \text{Eq. 2.19}$$

With s1, s2, s3, s4 defined it can be seen that they are the (estimated) slopes along the trajectory. S1 is the current slope corresponding on its actual time and x value. S2 and S3 are both estimates of the slope midway through the time step. S4 is the estimated slope at the end of the time step. The 4th order Runge-Kutta averages these slopes together with extra weight on the midpoint slopes s2 and s3. This 4th order Runge-Kutta process is run numerous times to calculate the actual vehicle state given the above equations of motion – the RK4() function calls the EOM2() function.

The equations of motion change slightly within the EOM2() function due to the presence of controls. In order to direct the vehicle in certain direction an LQR controller is implemented using Matlab's built in lqr() function. Appendix A shows the controller code which consists primarily of controls() function - and the implementation of the gain equation. The controls function is called early in the EOM2() function. It handles setup for Matlab's lqr() function. The lqr() function is the core of the

vehicle's control; documentation can be found on Matlab's official website (MathWorks 2017). The linear-quadratic regulator is a feedback controller.

The EOM2() function contains the call to the controls() function. The math that calculates the delta U (the change in the control vector from trim state) is as follows.

$$\Delta\vec{x} = \vec{x} - \vec{x}_{desired} \qquad \text{Eq. 2.20}$$

$$\Delta\vec{u} = -k\Delta\vec{x} \qquad \text{Eq. 2.21}$$

$$\vec{u} = \vec{u}_0 + \Delta\vec{u} \qquad \text{Eq. 2.22}$$

Where k is the gain constant returned from the controls() function. DeltaX is the difference between what is called the desired vehicle state and the current vehicle state. The state vector used for integration contains vehicle position, orientation/heading, velocities, and roll/pitch rates. The coordinates and heading are referenced from the inertial frame - not the body frame. The control vector u consists of the four thrust vectors and the heading rate. They are defined below:

$$\vec{x} = \begin{bmatrix} x \\ y \\ z \\ roll \\ pitch \\ heading \\ V_x \\ V_y \\ V_z \\ \dot{roll} \\ \dot{pitch} \end{bmatrix} \qquad \text{Eq. 2.23}$$

$$\vec{u} = \begin{bmatrix} T1 \\ T2 \\ T3 \\ T4 \\ \dot{heading} \end{bmatrix} \qquad \text{Eq. 2.24}$$

The control vector is used in the EOM2() function to compute the net force and torque as was described above in the equations of motion section. The above math is applied each integration time step in when the RK4() function called the EOM2() function.

The desired vehicle state is a vector of same length defining desired values for each variable. It is defined outside of the RK4() function and brought in as a global variable into EOM2() function. Some variables are less meaningful than others. For example - to command the vehicle forward in the x direction the following desired state vector will work.

$$\vec{x}_{desired} = \begin{bmatrix} x_{current} \\ y_{current} \\ z_{current} \\ 0 \\ 0 \\ 0 \\ +5(\frac{m}{s}) \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{Eq. 2.25}$$

The above desired state vector would continually be defined updating the x,y,z current positions and maintaining a positive five meters per second velocity in the x-direction. Effectively this tells the controller obtain level flight with the vehicle heading along the inertial frame's x-direction with a speed of 5 meters per second. Here the vehicle's position does not affect the controls and it tries to maintain zero velocity in the other directions with no roll or pitch after the five meters per second speed is obtained.

Additionally a desired x,y,z coordinate can be defined. The following desired position vector shows an example of this, however the time it takes the vehicle is much longer than defining a velocity, as the controller attempts to maintain the chosen velocities - which in this case is zero for each direction.

$$\vec{x}_{desired} = \begin{bmatrix} 5m \\ 10m \\ 6m \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{Eq. 2.26}$$

In order to address the slower motion when using the second method for controlling the vehicle position - a second layer of controls is introduced outside of the rk4() function. This layer of vehicle controls also plays a role in obstacle avoidance. The design of the second layer is to use a waypoint system. The waypoint system defines a X, Y, Z coordinate that vehicle will navigate to. Code is written to determine the unit vector representing the direction that the vehicle needs to travel in order to move to the desired waypoint. A desired magnitude of velocity is defined and used to determine the desired velocity to be used by the desired state vector. Often times the magnitude changes depending on the distance between the waypoint and the vehicle to help with more precise movement once the vehicle gets close to the waypoint. For example, by default the magnitude of the velocity is 5 meters per second, after the vehicle comes within 1m of the desired position, the velocity magnitude will be reduced to less than a meter per second to allow for more precise movement. This second layer allows for the implementation of a maneuver profile, where the profile used in the experiment is a two stage profile, when farther than .5m from the waypoint the max speed is 5 meters per second, else .5 meters per second. After $X_{desired}$ is defined the feedback controller takes over.

$$\hat{V}_{dir} = \frac{\vec{x}_{desired} - \vec{x}_{actual}}{|\vec{x}_{desired} - \vec{x}_{actual}|} \qquad \text{Eq. 2.27}$$

$$\vec{V}_{desired} = V_{mag}\hat{V}_{dir} \qquad \text{Eq. 2.28}$$

$$\vec{x}_{desired} = \begin{bmatrix} x_{current} \\ y_{current} \\ z_{current} \\ 0 \\ 0 \\ Head_{desired} \\ \vec{V}_{xdesired} \\ \vec{V}_{ydesired} \\ \vec{V}_{zdesired} \\ 0 \\ 0 \end{bmatrix} \qquad \text{Eq. 2.29}$$

In order for the feedback controller to work, a gain matrix k is needed. The gain matrix k is returned from the controls() function which is called each integration time step. The reason it is called each time step is because the gain k depends on the vehicle state itself. For example, if at the start of the vehicle's motion it is heading along the inertial frame's x-direction, a pitch down will cause a velocity in the x-direction – the gain matrix knows this. Later the vehicle could have a heading along the y-direction where a pitch down now causes a velocity in the y-direction. In order for the controller to work correctly the gain matrix k must be computed to know how the control vector changes the vehicle's state at the current state. The primary purpose of the controls() function is to set up for Matlab's lqr() function. The lqr() function requires four matrices, A, B, Q, and R. Controls() function numerically computes the Jacobean of the EOM() using numjac() (MathWorks 2017) function with respect to the state vector and the control vector. The returned Jacobian matrix is parsed and turned into two separate A and B matrices. Q and R are diagonal matrices.

$$EOM = \dot{\vec{x}}(\vec{x}, \vec{u}) \qquad \text{Eq. 2.30}$$

$$[A \quad B] = \left[\frac{\partial(EOM)}{\partial \vec{x}} \quad \frac{\partial(EOM)}{\partial \vec{u}}\right] \qquad \text{Eq. 2.31}$$

$$Q = \begin{bmatrix} 1/(x1max^2) & \cdots & 0 \\ \vdots & 1/(ximax^2) & \vdots \\ 0 & \cdots & 1/(x11max^2) \end{bmatrix} \qquad \text{Eq. 2.32}$$

$$R = \begin{bmatrix} 1/(u1max^2) & \cdots & 0 \\ \vdots & 1/(ximax^2) & \vdots \\ 0 & \cdots & 1/(u5max^2) \end{bmatrix} \qquad \text{Eq. 2.34}$$

The EOM() function is used here instead of EOM2() function in order to handle how numjac() function sends parameters. Here EOM represents the function that computes the time derivative of the state vector or x-dot. Also it should be noted that x-dot is a function of the state vector, x, and the control vector, u. The A, B, Q, and R matrices are passed to Matlab's lqr() function which returned the needed gain matrix k. It should be noted not much was done optimize the Q and R matrices, as there were even more important aspects to optimize in the overall project. There is room for improvement towards the controls of the vehicle, but the control aspect of the code works as well as is needed to achieve the goals of the project. What was needed was some form of autopilot that is able to navigate the vehicle in specified ways.

2.3 "THE WORLD" AND DETECTION

The world represents the true information present in the environment: objects, terrain, radioactivity levels, and any other detectable information. The real world is broken up into a 3-D array where each element represents a cube worth of space. The resolution of the cube (i.e. cube dimensions) can be specified when creating the world (and also has major impact on computation efficiency). The radioactive sources are placed into the environment where at any given time the source activity leads to a count rate which exhibits variation given by a Poisson distribution.

The spatial detection method seeks to mimic that of a 3-D laser range scanner much like a Hokuyo range scanner (Hokuyo Automatic Co. 2009). This method emits beams in multiple directions originating from the vehicle and searches the 3-D world array for occupied space. As for the radiation sensor, a simple Geiger counter model is used to account for the activity of a source or sources as a function of times and the distance from the aircraft mounted sensor. The counts are integrated when the EOM2() function is called.

The size, and form of “The World” can be very different. A few shape primitives are used in order to construct a more complicated environment. Appendix A shows many functions used to construct “The World”. When building a tree, a cylinder and a sphere can be combined to form a basic tree shape. Buildings, halls, floors, and ceiling can all be constructed using multiple blocks of differing shapes and sizes.

The build_world() functions generally define how “The World” will look along with its dimensions and resolution. These functions can vary on what parameters are sent as the dimension do not always need to be defined outside of the function. When using the build_world() function the most consistently required parameter would be the resolution. This variable often labeled “ppm” (pixel per meter) defines how large (or small) each cube will be. A small ppm like one will lead to a very blocky world where objects will always take up at least one meter cubed of space. If a larger ppm is used, then more objects can begin to fit in a single cubed meter of space. Ideally ppm would go towards infinite however due to memory and computation constraints a ppm of 5 is almost always used.

When building the world, a blank array full of zeros is first constructed. The size of the 3-D array depends on the size of the world and the world’s resolution. Separate functions like build_block() are called to place objects at specified locations with specified dimensions. These functions begin to fill in the world array with ones which represent occupancy of an object in that space.

In order to detect the object environment the detect_env3_imu() function is used. The detect_env3_imu() does more than simply detect, some additional operations it performs will be discussed later in the context of the estimation process. For now, it acts to mimic a laser range scanner. Figure 2.4 illustrates the general algorithm for detection in 2-D. Each beam is emitted at a given direction from the body x-axis defined by angles theta and phi. The algorithm systematically traces each beam line in the body frame. It then rotates that vector into the inertial frame, where the vehicle position vector is added. Based off the x, y, z coordinates of the beam at the current location the x, y, z

indices of the 3-D world array are then determined using the pixels (or cubes) per meter value. It will continue to trace the beam until one of three conditions are met. The three conditions are that it finds an occupancy, it goes out of bounds of "The World", or reached a defined max beam distance (usually set to five meters).

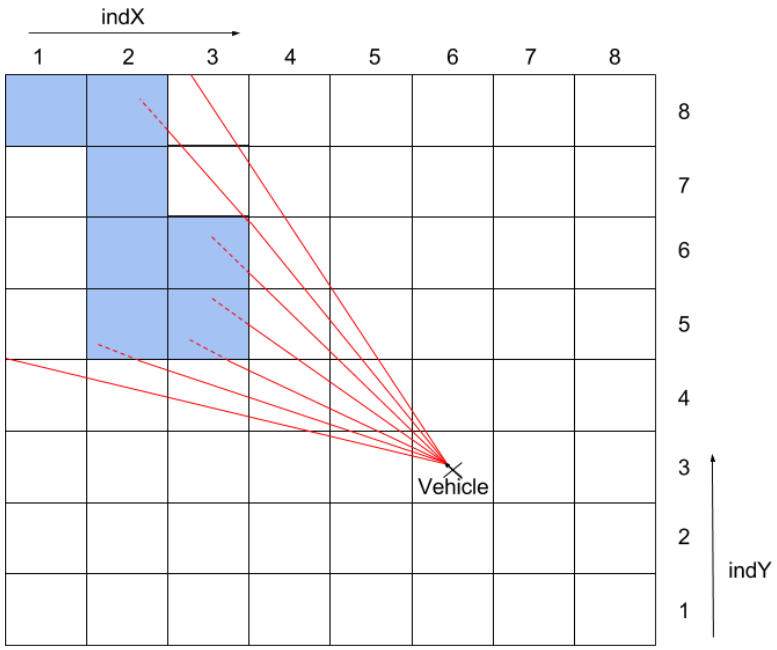

Figure 2.4 2-D Illustration of Obs. Detection: 2-D illustration of world cube occupancy showing obstacle sensing method. The blue blocks represent occupied space, solid red lines indicate sensor beam path before obstacle collision, dashed red lines represent beams measuring occupied space.

The next important part of "The World" is the representation of radioactive sources. Appendix A shows the functions used to place a source and determine the counts that are integrated each time step. In order to add a radioactive source into the environment the add_source() function is used an X, Y, Z coordinate and constant activity must be sent to the function. A struct keeps track of all sources, including their location and activity. The detection method again seeks to mimic a simple Geiger counter. It is know that the probability of a radioactive decay being detected follow a Poisson

distribution (Krane 1987). A Poisson distribution model is used to determine the number of counts recorded in a time step. Equations 2.35 and 2.36 show the math used to integrate the counts. Lambda is the average number of counts in a time step. It is determined from the activity A (in units of counts per second) and the time step deltaT. Lambda is required to determine the probability of k counts.

$$A = \frac{A_o}{r^2} \qquad \text{Eq. 2.35}$$

$$P(k) = \frac{\lambda^k e^{-k}}{k!} = \frac{(A*\Delta t)^k e^{-k}}{k!} \qquad \text{Eq. 2.36}$$

Once the probability of k counts is determined a random number generator is used to determine the number of counts k during a single integration time step. In order to help speed up computation time, and because for high values of k a Poisson distribution becomes a Gaussian, after, once k becomes greater than 16 counts the process is simplified to simply add Gaussian noise centered at the average lambda with a standard deviation equal to the square root of lambda.

$$\sigma = \sqrt{\lambda} \qquad \text{Eq. 2.37}$$

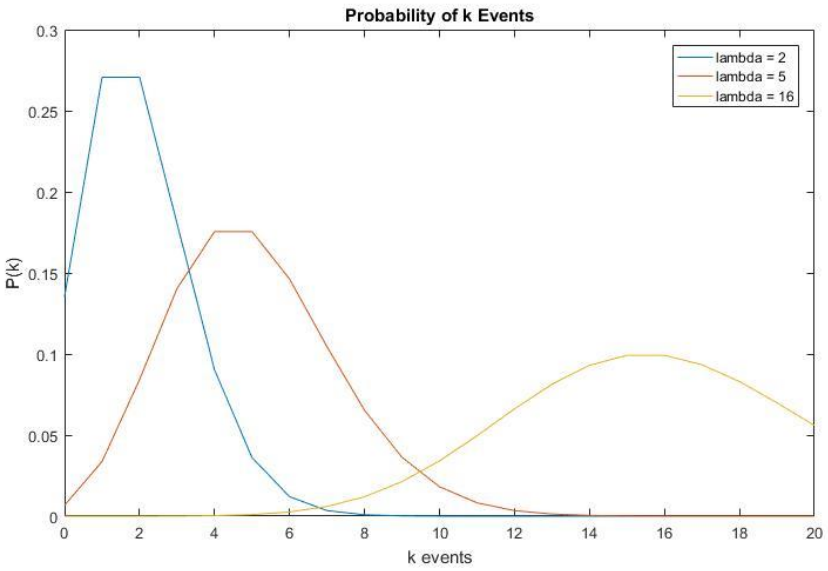

Figure 2.5 Poisson Distribution Graph: Graph displaying a Poisson Distribution for varying number of events. Generated in Matlab using the Poisson functions developed for radioactivity event counting.

Figure 2.5 shows the transition from a Poisson distribution to that of a Gaussian. This is done to show just how quickly the Poisson distribution turns into a Gaussian, supporting a transition to Gaussian noise when lambda is greater than 16. These counts are continually integrated and stored until a defined amount of time has passed before acquiring the data from the counter and clearing the counts for another count interval. The process of saving the radiation data and resetting the counts takes place outside of the integration process.

2.4 THE ESTIMATION PROCESS AND OBSTACLE AVOIDANCE

Up until this point the core of the project has been described. It is now time to discuss the butter, or the estimation process. There is no way to be 100% correct on where the vehicle is, its orientation, where the objects are, and the location of a radioactive source. It is assumed that GPS, compass, and IMU data can be acquired from the vehicle. The data from these sensors will provide a rough value for the vehicle position, heading, orientation and accelerations – however these

measurements are riddled with noise and will not be sufficient enough to estimate the vehicle state. Also available to us are measurement information from the range scanner and the Geiger counter. In order to better estimate the position of the vehicle and the state of the world a method is needed that combines an understanding of the physics and math behind the system and the information being received from the sensors. The method used is called the (Extended) Kalman Filter.

An introduction to both the Kalman and Extended Kalman Filter is first required before its specific implantation is discussed. The Kalman Filter is an algorithm that attempts to compute the minimum mean-square variance estimate of the state for a linear dynamical system (Haykin 2001). It is a recursive process suited well for applications via computer. A basic Kalman filter is a multi-step process. The explanation of the basic Kalman filter will mostly follow from Dr. Phil Kim's explanation (Kim and Huh 2011). Figure 2.6 drawn primarily from Kim's text, illustrates the Kalman filter algorithm.

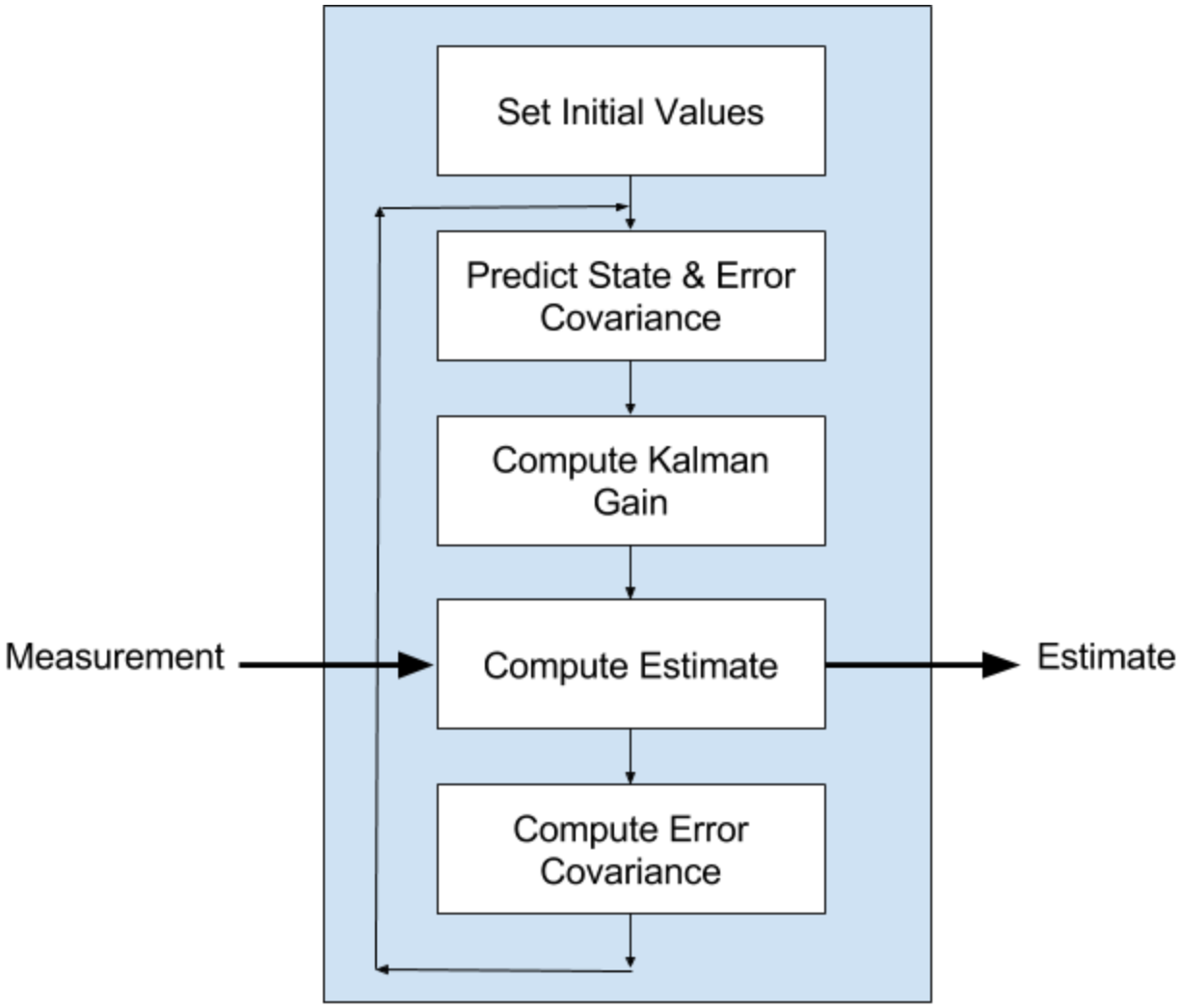

Figure 2.6 Illustration of KF Process: A visual illustration of the Kalman Filter process as taken from the text "Kalman Filter for Beginners: With MATLAB Examples".

The above flowchart shows the general flow of the Algorithm. Setting the initial values for x_0 and P_0 which are the initial state estimate and initial error covariance. These initial values are required as the process requires only information computed from the previous time step (k-1). The filter is broken up into two processes. The first is the prediction process which entail predicting the state and error covariance. The second is the estimation process estimation process which encompass computing

the Kalman gain, the estimate, and the error covariance. The system that is being estimated follows the following equations, showing how the process changes per time step.

$$x_{k+1} = Ax_k + Bu_k + w_k \qquad \text{Eq. 2.38}$$

$$z_k = Hx_k + v_k \qquad \text{Eq. 2.39}$$

Equation 2.38 shows that process model saying that the state variable in the next time step can be computed from the current time steps information plus some error which cannot be accounted for. It is assumed the error is Gaussian with zero mean. From equation 2.39 it should be seen that the measurements (z_k) can also be computed from the current state variable with some error. These will be defined in more detail later.

The general process for the Kalman filter is as follows: a prediction is made, then motion occurs and measurements are taken during that motion. Once that the motion has occurred, the Kalman gain is computed, and it is used to calculate the updated estimates and the updated error covariance. Equations 2.40 through 2.44 show the math behind the Kalman filter algorithm and Table 2.1 helps to label each variable from the equations.

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1} \qquad \text{Eq. 2.40}$$

$$P_k^- = AP_{k-1}A^T + Q \qquad \text{Eq. 2.41}$$

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1} \qquad \text{Eq. 2.42}$$

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-) \qquad \text{Eq. 2.43}$$

$$P_k = P_k^- - K_k H P_k^- \qquad \text{Eq. 2.44}$$

Table 2.1 KF Variables and Descriptions

Variable	Description	Size
$\hat{x}_k^-$	State Variable Prediction	Nx1 column vector
$\hat{x}_k$	State Variable Estimate	Nx1 column vector
u_{k-1}	Controls or Rate Vector	Lx1 column vector
z_k	Measurement Vector	Mx1 column vector
A	State Transition Matrix	NxN matrix
B	Controls/Rate Transition Matrix	NxL matrix
P_k^-	Predicted Error Covariance Matrix	NxN matrix
P_k	Corrected Error Covariance Matrix	NxN matrix
H	State-To-Measurement Matrix	MxN matrix
Q	Covariance Matrix of w_k	NxN matrix
R	Covariance Matrix of v_k	MxM matrix
w_k	Process Noise	Nx1 column vector
v_k	Measurement Noise	Mx1 column vector
K_k	Kalman Gain	NxM Matrix

To start, not only defining initial values for x_0 and P_0 but also the covariance matrices Q and R are required. A general first approach to defining these matrices is to define them as diagonal matrices using w_k and v_k respectively down the diagonals. w_k and v_k are the noise terms representing the error corresponding to the process model and measurements. w_k is comprised of the estimated standard deviation of our process model squared. v_k is comprised of the standard deviation corresponding to the devices used to make measurements.

$$w_k = \begin{bmatrix} {}_w\sigma_1^2 \\ {}_w\sigma_2^2 \\ \vdots \\ {}_w\sigma_N^2 \end{bmatrix} \qquad \text{Eq. 2.45}$$

$$v_k = \begin{bmatrix} {}_v\sigma_1^2 \\ {}_v\sigma_2^2 \\ \vdots \\ {}_v\sigma_M^2 \end{bmatrix} \qquad \text{Eq. 2.46}$$

The Covariance Matrix *Q* is somewhat notorious for being difficult to accurately build as often the standard deviation of the process model is not known. It typically requires tweaking to fine tune Kalman filter performance. However, a starting approach to building it is to put the "guessed" estimated standard deviations down the diagonal as shown.

$$Q = \begin{bmatrix} {}_w\sigma_1^2 & \cdots & 0 \\ \vdots & {}_w\sigma_i^2 & \vdots \\ 0 & \cdots & {}_w\sigma_N^2 \end{bmatrix} \qquad \text{Eq 2.47}$$

The *R* matrix is done in a similar manner.

$$R = \begin{bmatrix} {}_v\sigma_1^2 & \cdots & 0 \\ \vdots & {}_v\sigma_i^2 & \vdots \\ 0 & \cdots & {}_v\sigma_M^2 \end{bmatrix} \qquad \text{Eq. 2.48}$$

To reiterate, often times ${}_w\sigma_i^2$ is unknown or ambiguous, and a reasonable guess for the value is used as a starting point. Constructing A, B, and H can vary in difficulty depending on the system being estimated; they represent the model for the state transition and the state to measurement matrices. A detailed example will be shown below when the specific application is discussed. A simple example of the state to measurement matrix H however can be quickly discussed. Let's assume that our state vector x has a two state variables, i.e. its x-position and x-velocity. Let's then also assume that we have a device that directly measures its x-position with some error v_k. Additionally, say the acceleration is also known experimentally from an accelerometer. In this example, A would be a two by two matrix.

We will assign the known acceleration value to the u vector which would be of length one. The B matrix in this application would be a two by one matrix. A, B, x, and u would satisfy equation 2.38. H would be a one by two matrix so that z_k vector has length of one and follows equation 2.39. For this example; A, B, and H look as follows:

$$A = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \qquad \text{Eq. 2.49}$$

$$B = \begin{bmatrix} .5\Delta t^2 \\ \Delta t \end{bmatrix} \qquad \text{Eq. 2.50}$$

$$H = \begin{bmatrix} 1 & 0 \end{bmatrix} \qquad \text{Eq. 2.51}$$

Given x to be:

$$\vec{x} = \begin{bmatrix} x \\ v \end{bmatrix} \qquad \text{Eq. 2.52}$$

$$\vec{u} = [a] \qquad \text{Eq. 2.53}$$

And applying these to equations 2.40 we begin to form the kinematic equations.

$$x_{k+1} = x + v\Delta t + a(.5\Delta t^2) \qquad \text{Eq. 2.54}$$

$$v_{k+1} = v + a\Delta t \qquad \text{Eq. 2.55}$$

In some implementations of the Kalman filter vector u and matrix B may not apply to their system and may not exist in the process model. Above is a description of the basic Kalman filter and a simple example of a linear dynamic system is shown. In some cases, the system is not easily linearized about a single value – i.e. what if it was proportional to the velocity or acceleration squared. Such is the case in the specific application of the Kalman filter here. In order to handle this type of non-linear system, the first go to is what is called the Extended Kalman Filter.

There is one major change mathematically from the basic Kalman filter to the Extended. Instead of equations 2.38 and 2.40, they become equations 2.56 and 5.57 respectively.

$$x_{k+1} = f(x_k, u_k) + w_k \qquad \text{Eq. 2.56}$$

$$\hat{x}_k^- = f(x_{k-1}, u_{k-1}) \qquad \text{Eq. 2.57}$$

This can also be applied to the measurements as well changing equation 2.39 into equation 2.58.

$$z_k = h(x_k) + v_k \qquad \text{Eq. 2.58}$$

It is now apparent that rather than a simple linear relationship defined through matrices, the relationships are expressed in functions f and h. In the case of a simple linear relationship it can be simplified back to the basic Kalman Filter. However – the Kalman filter assumes a linear dynamic system. This means that the system needs to be linearized as well as possible around the current state values. This effectively changes how A, B, and H are formed. A quick inspection shows no dependence on the matrix B for the Kalman filter process, it is a matrix that can be ignored in the Extended Kalman filter. Like above with the LQR process, the solution to getting A, and H matrices lies in the Jacobian.

$$A = \left[\frac{\partial f(x_{k-1}, u_{k-1})}{\partial \hat{x}_k}\right] \qquad \text{Eq. 2.59}$$

$$H = \left[\frac{\partial h(x_{k-1}, u_{k-1})}{\partial \hat{x}_k^-}\right] \qquad \text{Eq. 2.60}$$

By taking the Jacobian the system is effectively linearized about the state estimate as opposed to the trim state. These A and H matrices are computed at each time-step for the EKF. There may be cases where the state estimates experience little to no change where perhaps the matrices do not need to be recomputed. Once the A and H matrices are acquired, the Kalman filter algorithm can be followed just as defined above.

Once an estimated world and vehicle position is produced, it is used to check for obstacles that the vehicle must avoid. A rough estimate of the vehicle's flight path is computed as a line from the vehicle to its desired waypoint. The direction of the line from the vehicle is taken from the velocity unit vector computation shown in equation 2.27. This line is checked incrementally starting at the vehicle location towards the final destination. If an obstacle is detected in the estimated world, a flag is triggered causing the gradient of the obstacles to be computed at the collision point. With that gradient

a new interim waypoint is determined and takes over as the current waypoint. This process is repeated every time step and will replace newer interim waypoints with the current interim waypoint if another potential collision is detected. If the routine deems it safe to navigate to the desired waypoint it will then remove the interim waypoint and begin navigating to the desired position. The waypoint navigation is handled through the second layer of controls introduced above.

2.5 SPECIFIC APPLICATIONS OF THE KALMAN FILTER

There are three implementations of the Kalman Filter used. They will be discussed in the order of their development. The first implementation estimates the vehicle state, using IMU, compass and GPS readings as vehicle state measurements. Additionally, it estimates a static 10-meter by 10-meter by 10-meter world. The vehicle is free to move anywhere in this 1000-meter cubed space. Measurements of the space are taken from a simulated range finder type of sensor similar to a Hokuyo range finder or X-Box Kinect. Radiation measurements are taken at the vehicle location and count as measurements toward an estimated radiation reading corresponding to a specific location in the world.

The second filter application is similar to the first except the world being estimated is no longer static. It fixes the vehicle at the center of the 1000-metter cubed space. As the vehicle moves it attempts to estimate the transition of information throughout the space as the vehicle moves. For example, if there is an object 2m ahead of the vehicle and in the next time step the vehicle has moved forward by 1m, the information of the object should transition to being in the cube 1m in front of the vehicle. Radiation information is handled in a similar way, as the vehicle moves the information is transitioned in a corresponding manner. Due to non-linear elements of the transition model, an Extended Kalman filter is used

The third Kalman filter no longer estimates the space around the vehicle but instead it estimates the location of measured landmarks, the location of a single radioactive source, and the source's intensity. Again due to the non-linear elements of the transition model, an Extended Kalman filter is

used here as well. This method borrows some elements from Dissanayake (Dissanayake, Newman et al. 2001) regarding landmark identification process.

2.5.1 KALMAN FILTER APPLICATION 1

Version 4.2 of the code hosts the first Kalman Filter implementation, which works to estimate the vehicle position, vehicle orientation, and a static world around the vehicle. This implementation is a standard Kalman Filter. The estimated world is fixed and of finite size. This means the space the Kalman filter attempts to estimate does not move relative to the vehicle. The space is limited to a 10 meter by 10 meter by 10 meter volume. This volume is broken up into 1000 1 meter cubed blocks. Two values are estimated for each block - the first is occupancy (i.e. there is some object measured in that location) and the second is a radiation reading measured at the location of that block. The first version is the simplest of the three Kalman Filters used.

The vehicle state variables estimated are the x,y, and z position in the inertial frame, the vehicle roll, pitch and heading from the inertial x-axis, and the vehicle's inertial x,y, and z velocities. These variables make up the vehicle portion of the state vector $\vec{x}$. The obstacle portion of the state vector are 1000 variables, each variable represents the occupancy of the n^{th} cube. The value can range from negative to positive, where the greater the positive number the greater the certainty of an obstacle being present in the cube. The next 1000 variables represent radiation measurements at corresponding cubes. The vehicle does not enter every cube, cubes with no measurement values contain a default value of zero. The total estimated state vector for this model is shown below, with $\vec{x}_{Veh_k}$ being the vehicle state variables, $\vec{O}_k$being the cube occupancy state variables, and $\vec{R}_k$ being the radiation state variables.

$$\vec{x}_k = \begin{bmatrix} \vec{x}_{Veh_k} \\ \vec{O}_k \\ \vec{R}_k \end{bmatrix} \qquad \text{Eq. 2.61}$$

Figure 2.7 attempts to illustrate the Kalman Filter design. In the left sector, is the environment broken up into boxes which will represent a 2-D slice of the KF cube world. The black arrow denotes the location and heading of the vehicle. The blue box represents some object present in the environment. Following the purple arrow to the right, this is the same world represented fully in through the KF variables. Barring error in measurements and estimates, the KF estimation/measurement process would identify cube that the vehicle is when it made a radiation measurement, shown by the green x. The obstacles would be found to be in the two cubes with the red circles, the measurement are based off of the vehicle state (both position and orientation estimates) in order to determine where the obstacle measurement was made. It is important to note that the vehicle position/orientation estimates may be exact down to some decimal value, however the cubes represent a range of position values. This can also be seen when looking at the obstacle estimate/measurement where the KF value corresponding to the left red circle would symbolize occupancy regardless of less than half of the space actually being occupied, as seen by the blue object itself. The actual numerical value that the variables hold may differ, where the one of the left may be less than the one on the right, due to more positive obstacle measurements being made within that cube. The transition and measurement models will be discussed further - however a base understanding of the design is essential to understanding them.

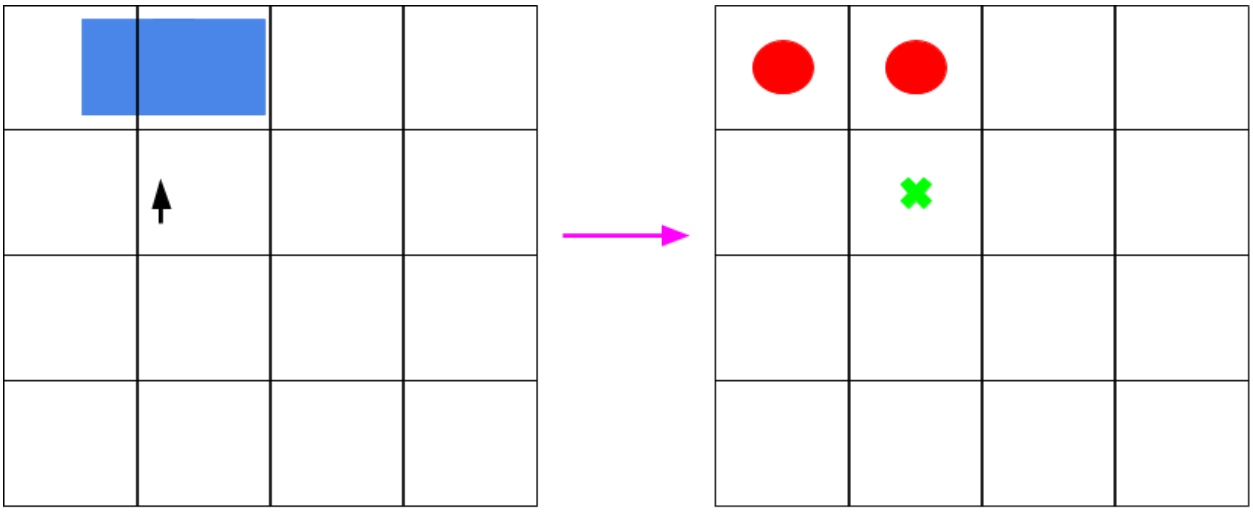

Figure 2.7 2-D Illustration of KF1 Occupancy and Radiation Data Representation: A visual illustration of the KF1 algorithm as it discovers occupied space and estimates occupancy. The blocks represent the space the KF is estimating occupancy of. On the left the object is represented by the blue, the vehicle by the arrow. On the right the KF estimates the vehicle to be in the block with the green "x" and estimates an occupancy value of the two measured blocks.

2.5.1.1 THE TRANSITION MODEL

The transition model has three parts, the vehicle model, the world occupancy model, and the radiation model. For the vehicle model, the motion can be modeled by the following equation.

$$\vec{x}_{Veh_{k+1}} = A_{veh}\vec{x}_{Veh_k} + B_{veh}\vec{u}_k + \vec{w}_{k+1} \qquad \text{Eq. 2.62}$$

A_{veh}: the state transition matrix

B_{veh}: the rate transition matrix

$\vec{u}_k$: rate vector representing IMU and gyroscope measurements

$\vec{w}_{k+1}$: uncorrelated process noise with zero mean and covariance Q

Equation number 2.63 shows the vehicle state x at some time step $k+1$.

$$\vec{x}_{Veh_k} = \begin{bmatrix} x \\ y \\ z \\ roll \\ pitch \\ heading \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \qquad \text{Eq. 2.63}$$

u_k is a rate vector representing IMU and gyroscope measurements. The IMU vector is rotated into the inertial frame and is shown below.

$$\vec{u}_k = \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \\ \dot{roll} \\ \dot{pitch} \\ \dot{heading} \end{bmatrix} \qquad \text{Eq. 2.64}$$

Following simple kinematic motion, the vehicle transition model equation for the y-direction is shown in equations 2.65 and 2.66. The same is true for the x and z directions. The transition model for the roll, pitch, and heading are also shown in the example equation 2.67 for the vehicle roll, as pitch and heading transition equations are identical. It is important to remember, for this application, that heading is defined as the angle from the body x-axis to the inertial x-axis. It is assumed that the gyro measurements provide a good enough heading rate to be used by the KF.

$$y_{k+1} = y_k + \dot{y}_k \Delta t + \frac{1}{2}\ddot{y}_k \Delta t^2 \qquad \text{Eq. 2.65}$$

$$\dot{y}_k = \ddot{y}_k \Delta t \qquad \text{Eq. 2.66}$$

$$roll_{k+1} = roll_k + \dot{roll}_k * \Delta t \qquad \text{Eq. 2.67}$$

The above equations show how each vehicle state transitions during each type step. In order for the model defined by equation 2.40 the A_{veh} and B_{veh} matrices have to be defined as follows.

$$A_{veh} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{Eq. 2.68}$$

$$B_{veh} = \begin{bmatrix} \frac{\Delta t^2}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{\Delta t^2}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\Delta t^2}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 0 & 0 & \Delta t \\ \Delta t & 0 & 0 & 0 & 0 & 0 \\ 0 & \Delta t & 0 & 0 & 0 & 0 \\ 0 & 0 & \Delta t & 0 & 0 & 0 \end{bmatrix} \qquad \text{Eq. 2.69}$$

By combining the equations 2.68 and 2.69 into the following model equation, the above kinematic equations are acquired for the vehicle transition model.

$$\vec{x}_{Veh_{k+1}} = A_{veh}\vec{x}_{Veh_k} + B_{veh}\vec{u}_k \qquad \text{Eq. 2.70}$$

The transition model for the environmental obstacle estimation is rather simple. It is assumed that the environment is static, therefore the obstacle status from one time step to the next should be the same. This makes the transition model trivial.

$$\vec{O}_{k+1} = \vec{O}_k \qquad \text{Eq. 2.71}$$

However in order to follow the KF filter model we must determine a state transition matrix A_O for the obstacle estimation process.

$$\vec{O}_{k+1} = A_O\vec{O}_k \qquad \text{Eq. 2.72}$$

In order for equation 2.72 to satisfy equation 2.71 A_O would have to be an identity matrix.

$$\vec{O}_{k+1} = \boldsymbol{I}_O\vec{O}_k \qquad \text{Eq. 2.73}$$

Where $\boldsymbol{I}_O$ would be 1000 by 1000 identity matrix where $\vec{O}_k$ is of length 1000.

The transition model for the radiation estimation behaviors similarly. The source(s) are assumed to be static - so all measurements barring statistical noise should be the same. That means the transition model is identical to the obstacle model.

$$\vec{R}_{k+1} = \vec{R}_k \qquad \text{Eq. 2.74}$$

Again in order to follow the KF model a state transition matrix A_R must again be determined.

$$\vec{R}_{k+1} = A_R\vec{R}_k \qquad \text{Eq. 2.75}$$

Once again, in order for equation 275 to satisfy equation 2.74 A_R must be a 1000 by 1000 identity matrix.

$$\vec{R}_{k+1} = \boldsymbol{I}_R\vec{R}_k \qquad \text{Eq. 2.76}$$

$$A = \begin{bmatrix} A_{veh} & 0 & 0 \\ 0 & \boldsymbol{I}_O & 0 \\ 0 & 0 & \boldsymbol{I}_R \end{bmatrix} \qquad \text{Eq. 2.77}$$

$$B = \begin{bmatrix} B_{veh} \\ \vdots \\ 0 \end{bmatrix} \qquad \text{Eq. 2.78}$$

The final state transition matrix A, and rate transition matrix B are above. The state transition matrix A will be a 2009 by 2009 matrix. The rate transition matrix B will be a 2009 by 6 matrix. The dimensions of the matrices are essential and in order to do the computation the dimensions must be compatible with the state and rate vectors to return a vector of equal length to the state vector - this fact is more important later on during other KF implementations.

2.5.1.2 THE MEASUREMENT MODEL

The measurement model again has three parts to it. There are measurements corresponding to the vehicle state, measurements corresponding to obstacle locations, and radiation measurements from the Geiger counter. The measurement vector is of length 1007. The first six variables correspond to the vehicle state (GPS X, Y, Z coordinate and roll, pitch and heading measurements). The last variable

represents the radiation measurement that is made. The 1000 variables left correspond to obstacle measurements in the 1000 cube space that the KF is attempting to estimate.

$$\vec{z} = \begin{bmatrix} \vec{z}_{Veh} \\ \vec{z}_{Obs} \\ z_{Rad} \end{bmatrix} \qquad \text{Eq. 2.79}$$

Some things are important to note. Not every cube is measured in every iteration of the KF. Additionally, a radiation measurement is not made in every iteration. Looking back at the KF introduction, the state-to-measurement matrix H is applied onto the state vector x in order to acquire the measurement vector. The following equation shows the general observation model equation. The measurements made are a function of x, however there are is non-correlated zero mean noise v_k present in the measurements

$$\vec{z} = H\vec{x} + \vec{v}_k \qquad \text{Eq. 2.80}$$

In order to handle different cubes being measured at different iterations, and the frequency of radiation measurements being different from that of vehicle state and obstacles measurements, a dynamic state-to-measurement matrix is required. The state-to-measurement matrix can be broken up again into three portions corresponding to the different types of measurements being made.

$$H = \begin{bmatrix} H_{Veh} & \dots & 0 \\ \vdots & H_{Obs} & \vdots \\ 0 & \dots & H_{Rad} \end{bmatrix} \qquad \text{Eq. 2.81}$$

Since vehicle state measurements are made at the same frequency of the KF computation the H_{Veh} matrix is constant. It says that the vehicle, x, y, z, roll, pitch, and heading state variables are being measured. The H_{Veh} matrix looks as follows.

$$H_{Veh} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \qquad \text{Eq. 2.82}$$

Multiplying the H_{Veh} matrix with a vector of the first 9 variables from the state vector, a vector of length 6 would be returned with the values corresponding to the vehicle x, y, z, roll, pitch, and heading.

The H_{Obs} matrix is the most dynamic/largest portion of the state-to-measurement matrix. A very important fact about the state-to-measurement matrix is that it tells the Kalman Filter what states are being measured. This is the portion of the state-to-measurement matrix that will pull out from the state vector which cubes have been measured. The state-to-measurement matrix will have a direct effect on the Kalman Gain and State Variable Estimate. In his model, it will be predicted that what is measured will be the estimated states. For example, if cube 256 is measured, the H_{Obs} matrix element in row 256 and column 256 will be set to 1. This makes it so that the estimated measurement will be equal to the current value in cube 256's obstacle state variable. To achieve this, during the measurement process, the cubes of which the laser range finder beams pass through/measure in are kept track of. If a positive measurement is made in some cube, then a positive value of 200 is added to that measurement vector variable corresponding to it. After a positive measurement is made, the search algorithm along that beam stops, going to the next. This means that there are far more negative (no objects found) results from the search algorithm. In order to clear false positives in the obstacle measurements, a negative value is added to the corresponding measurement vector variable for the respective cubes. Due to many more negative results during the search algorithm, a smaller negative number of -.1 is added where a single positive could be as high as plus 5. This means that it will take many negative results to counteract a positive result. This is both a strength and a weakness.

The H_{Obs} matrix is a diagonal matrix, only the diagonal elements can potentially take a non-zero value, all other values will be zero. During the function responsible for detecting obstacles, the cubes that measurements are made in are kept track of in order to properly populate the real measurement matrix, and correctly construct the H_{Obs} matrix. The functions responsible for handling obstacle

detection and forming the actual measurement vector for this version are detect_env3_imu() and handle_Measurements_imu() and can be seen in the appendix.

The H_{Rad} is a 1000 by 1000 matrix of all zeros. For the majority of the simulations the frequency of radiation readings is one second. The KF frequency is ten iterations per second. This means that nine out of every 10 KF iterations the H_{Rad} values will be zero. When a radiation reading occurs, the cube that the vehicle is in during the reading is determined. The value that corresponds to this cube is changed to a one in order to tell the KF that the radiation measurement was made and in what cube it was made in. The final function that that forms the entire state-to-measurement matrix is build_imuHk(). It is sent the information to form each of the three matrices and puts them into a final state-to-measurement matrix used by the KF.

2.5.1.3 THE KALMAN FILTER INITIALIZATION

In order to begin the Kalman Filter process, a few matrices must be initialized. These matrices are the covariance matrices corresponding to the state process noise and measurement noise Q and R respectively. In this version these matrices are built using the two functions build_imuEx() and build_imuEz(), where Ex corresponds to Q and Ez corresponds to R. Looking at equation 2.47 and 2.48 for Q and R above, the sigmas used to form them reference the noise from the estimation and measurement models $\vec{w}_k$ and $\vec{v}_k$. The $\vec{w}_k$ values correspond to an estimate of what the error in the model may be. When building the Q matrix, the sigma values can change – however for most applications shown in the results, the Q matrix is shown below in the following equations and table.

$$Q = \begin{bmatrix} Q_{Veh} & \cdots & 0 \\ \vdots & Q_{Obs} & \vdots \\ 0 & \cdots & Q_{Rad} \end{bmatrix} \qquad \text{Eq. 2.83}$$

$$Q_{Veh} = \begin{bmatrix} {}_{x}\sigma^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & {}_{y}\sigma^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & {}_{z}\sigma_1^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & {}_{roll}\sigma^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & {}_{pitch}\sigma^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & {}_{head}\sigma^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & {}_{Vx}\sigma^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & {}_{Vy}\sigma^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & {}_{Vz}\sigma^2 \end{bmatrix} \quad \text{Eq. 2.84}$$

$$Q_{Obs} = \begin{bmatrix} {}_{Obs}Q_1 & \cdots & 0 \\ \vdots & {}_{Obs}Q_i & \vdots \\ 0 & \cdots & {}_{Obs}Q_{1000} \end{bmatrix} \quad \text{Eq. 2.85}$$

$$Q_{Rad} = \begin{bmatrix} {}_{Rad}Q_1 & \cdots & 0 \\ \vdots & {}_{Rad}Q_i & \vdots \\ 0 & \cdots & {}_{Rad}Q_{1000} \end{bmatrix} \quad \text{Eq. 2.86}$$

Table 2.2 Typical Sigma Values for Q Matrix

Variable	Value
$_{x}\sigma$	.02m
$_{y}\sigma$	.02m
$_{z}\sigma_1$	.02m
$_{roll}\sigma$	.1rad
$_{pitch}\sigma$	.1rad
$_{head}\sigma$	.1rad
$_{Vx}\sigma$	.2m/s
$_{Vy}\sigma$	.2m/s
$_{Vz}\sigma$	.2m/s
$_{Obs}Q_i$	5
$_{Rad}Q_1$	50

For the R matrix, these sigmas are far more straight forward in selecting. These sigmas represent roughly the measured error in the measurements themselves. Since the standard deviations for the measurements are an adjustable parameter, the standard deviations used to generate error are passed directly into the build_imuEz() function. The values of error vary, however the GPS x, y, z measurement standard deviations are as a default set to 1m. The compass error is set to .2 radians. The object measurement error can vary as error comes from error in the vehicle position, measurements, additionally measured values can vary as the number of position, and negative hits may not be the same as the vehicle moves. A value of 40m is used to build the R matrix. This value can vary but for most examples it takes a value of 40m. The radiation measurement error is set at 1 counts, this

value is not realistic, in reality the error generally takes a value of about the square root of the counts, but the counts can drastically vary from distance to source and source intensity. Again, this can vary but for most runs it is set to 1 counts for the construction of the R matrix.

The final matrix that requires some initialization before the KF process can begin is the initial Covariance Matrix P_0. This matrix says a lot to the KF about what is initially known. In almost all runs the Covariance Matrix is set to zero. However in some runs a value of 1 is used in the diagonal for both obstacle and radiation estimates. A zero covariance matrix tends to weigh current information over new measurements while a greater covariance matrix will do the opposite, weighing new measurements higher. As the KF process progresses so does the covariance matrix.

2.5.1.4 ADDITIONAL HELPER FUNCTIONS

Some additional functions are used to aid in the process of the KF. In this version handle_Measurements_imu() (APPENDIX A) is used to form the measurement vector $\vec{z}$. This function is sent all the measurement information and ensures that the error is added to each measurement returning a vector corresponding to each measurement.

The find_measurement_index2_imu() function (APPENDIX A) returns the index of the cube that the specified x,y,z coordinate falls into. It is used to find which cube an obstacle and radiation measurement is made in.

The get_xyz_from_state_imu() function (APPENDIX A) is a function that breaks down the state vector $\vec{x}_k$ into a more usable form. It returns the x,y,z coordinate and value of each cube with occupied data greater than zero. It also does the same for cubes with radiation measurements and estimates. The output of this function is mostly used for visual output and obstacle avoidance.

2.5.2 KALMAN FILTER APPLICATION 2

Version 6 is the second KF application. This version uses an Extended Kalman Filter and attempts to address both the finite static world issue, and computation time of the first. The design

retains similar components to the first KF implementation. The EKF estimates two sets of information of 125 cubes. Instead of the fixed cubes in the inertial frame they are now fixed on the vehicle, with the vehicle in the center of the space. The new cube size is also different. They have a length, width, and height of 2 meters. The resolution of the estimated world is halved. With the estimated world fixed about the vehicle now, the information must transfer from one cube to another as the vehicle moves. Figure 2.8 attempts to illustrate the core design to the second EKF. The vehicle is represented by the arrow in the center of the estimated Kalman Filter world. At one instant on the left, there is obstacle or radiation information in a single cube as represented by the red dot. The occupancy is represented by a number that ranges from negative (no occupancy) to positive (occupancy). Much like the previous KF application the greater the positive number the more likely the cube is occupied; this number can be referred to as a confidence level. As the vehicle begins to travel forward, it displaces by some deltaX amount. Depending on the deltaX, the greater the amount of information will be transferred and averaged into the adjacent cubes. This can be seen on the right side of the figure where the size of the original circle has reduced, while a new smaller circle has been formed. For example, with cube sizes of two meters, if the vehicle displaced one meter in a single time step, half of the information or half of the confidence level will be averaged with what was already present. This is a simplified example of the estimation model for the filter, it is important to have an overview before diving into the math behind the model. Due to the nature of the estimation model, an Extended Kalman Filter (EKF) is used.

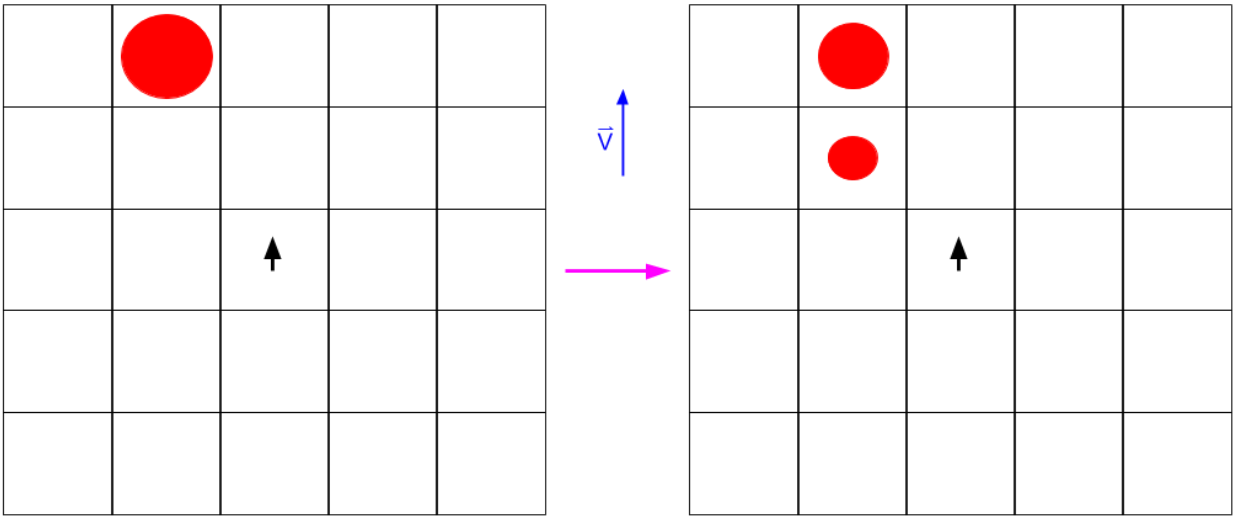

Figure 2.8 2-D Illustration of EKF 2 Data Transition Model: An illustration of EKF2's world model and how the information transitions during motion. On the left the information is as represented in an instant. The vehicle is at the center of the estimated space as represented by the arrow, some occupancy is present as seen by the red circle. As the vehicle moves forward, the information (both occupancy and radiation information) transitions. The transition is seen as some information is lost in the previous block, but new information emerges in a closer block.

2.5.2.1 THE TRANSITION MODEL

The transition model makes use of a function to define the estimate.

$$\vec{x}_{k+1} = f(\vec{x}_k, \vec{u}_k) + \vec{w}_{k+1} \qquad \text{Eq. 2.87}$$

Two versions of the function $f(\vec{x}_k, \vec{u}_k)$ can be found in version 6 and are below (APPENDIX A). Respectively the first is used as the primary function for the EKF's estimate f1(). The second, f2(), is a slightly modified version which gives identical output but uses an input that is more compatible with the numjac() function. This is because in order to acquire the State Transition Matrix A, the Jacobian of the function $f(\vec{x}_k, \vec{u}_k)$ is numerically computed. What the function does is use the estimated amount of vehicle motion $\Delta\vec{x}$ to determine how much and in what direction the information will transition. In the process of calculating the vehicle state transition $\Delta\vec{x}$ is computed.

$$\Delta\vec{x} = +\dot{\vec{x}}_k \Delta t + \frac{1}{2}\ddot{\vec{x}}_k \Delta t^2 \qquad \text{Eq. 2.88}$$

The informational transition is in the opposite direction of the vehicle motion. Once the $\Delta\vec{x}$ is determined and the function decides which direction the transition will be in, it uses a weighted average based on how far the vehicle moved and the length of the cubes, which are set to two meters. Figure 2.9 attempts to illustrate the information matrix B, and the displacement vector in two dimensions.

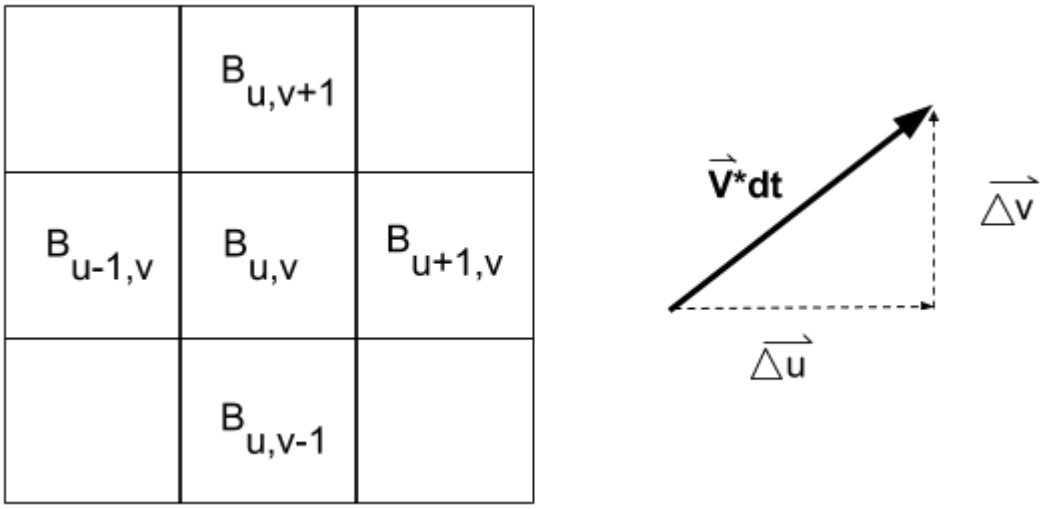

Figure 2.9 Illustration of EKF 2 Data Transition Model: An illustration giving further context to the transition model, where information in the $B_{u,v}$ block is being transitioned based on a displacement vector $\vec{V}dt$ on the right.

Here, the vehicle transitions in the positive u, and v directions. This means that the information in $B_{u,v}$ will be averaged with information coming from $B_{u+1,v}$ and $B_{u,v+1}$. It is a weighted average and this weighted average depends on the size of the vehicles estimated displacement. The math behind the transition model contains the third w dimension.

$$B_{new} = \left[\frac{1}{2m}\left(\left|\overrightarrow{\Delta u}\right|B_{u+1,v,w} + \left(1 - \left|\overrightarrow{\Delta u}\right|\right)B_{u,v,w}\right) + \frac{1}{2m}\left(\left|\overrightarrow{\Delta v}\right|B_{u,v+1,w} + \left(1 - \left|\overrightarrow{\Delta v}\right|\right)B_{u,v,w}\right) + \frac{1}{2m}\left(\left|\overrightarrow{\Delta w}\right|B_{u,v,w+1} + \left(1 - \left|\overrightarrow{\Delta w}\right|\right)B_{u,v,w}\right)\right]\frac{1}{3} \qquad \text{Eq. 2.89}$$

$$B_{new} = \frac{1}{6m}\left[\left|\overrightarrow{\Delta u}\right|B_{u+1,v,w} + \left|\overrightarrow{\Delta v}\right|B_{u,v+1,w} + \left|\overrightarrow{\Delta w}\right|B_{u,v,w+1} + B_{u,v,w}(3 - \left|\overrightarrow{\Delta u}\right| - \left|\overrightarrow{\Delta v}\right| - \left|\overrightarrow{\Delta w}\right|)\right] \quad \text{Eq. 2.90}$$

$B_{u,v,w}$ represents both spatial and radiation information. The transition model for the vehicle state is identical to the above KF. The function f1 does this for each value of u, v, and w. When the indices are at the edge, new information to be averaged is set at a default value of zero. The idea behind this model is that the environmental information follows the vehicle around. As the vehicle moves, so does the location of the space that the EKF attempts to estimate.

2.5.2.2 THE MEASUREMENT MODEL

The measurement model is nearly identical to the first Kalman Filter. It does not use a function to compute the measurement estimate, instead uses a state-to-measurement matrix H. GPS and gyro data is taken as measurements for the vehicle state. The proximity observation model is the same, as the range scanner looks for obstacles both positive and negative values add up to represent occupancy vs non-occupancy of the different EKF cubes that are being measured. Where it differs slightly is in the radiation model. Now because the EKF world follows the vehicle around the radiation measurement can only be made in the same cube every time, which is the cube the vehicle is always in. This means the only value in the H_{Rad} matrix that can change to 1 corresponds to the cube that the vehicle is always in, which is the 63rd cube.

2.5.2.3 THE EXTENDED KALMAN FILTER INITIALIZATION

Due to extreme similarities in the second EKF model and the first, the initialization process is very nearly identical. The size of the Q and R matrices differ due to different number of states and measurements. The same builder functions, build_imuEx() and build_imuEz(), are used to form the matrices, the errors and standard deviations are predefined and sent to the functions. Most of the error values for both Ez and Ex are similar if not the same as the previous application. The results section will point out any changes in values. The P_0 covariance matrix is also initialized the same as the first KF design as well.

2.5.3 KALMAN FILTER APPLICATION 3

The third design again uses an Extended Kalman Filter. It takes as step back from the previous designs and drastically changes the world and radiation model of the filter. It borrows from Dissanayake (Dissanayake, Newman et al. 2001), applying a landmark model of the world. The idea is that obstacle in the world can be represented as a landmark which has an x, y, z coordinate. Measurements made within a certain proximity of these landmarks can be perceived as a measurement of the landmarks and used to correct their location. This idea was applied in Dissanayake in both experiment and simulation in 2-dimensions. Now instead of estimating spaces of the world that drone flies in, the new EKF attempts to estimate objects instead. The number of landmarks is dynamic and at the start of the simulation the number of landmarks is at zero until measurements begin to be made. This will be talked about more in the transition and measurement model section.

The other major change is to the radiation model. Instead of estimating measurements, the new EKF model attempts to localize a single radioactive point source in the environment. To do this it estimates not only its position but also the strength or intensity of the source. This drastically reduces the number of variables that the radiation model estimates down to four.

2.5.3.1 THE TRANSITION MODEL

Despite drastic changes in the design the transition model does not differ from the original KF.

$$\vec{x}_{k+1} = f(\vec{x}_k, \vec{u}_k) + \vec{w}_{k+1} \qquad \text{Eq. 2.91}$$

The function used to define the transition model is called TransitionModel() seen in appendix A. Mathematically this function uses the exact same equations as defined for the vehicle state portion of the transition model in the first KF design. Simple kinematic motion is described for the vehicle state. The landmark transition model is also identical to the original KF. The landmarks being estimated are assumed to be static, so the estimates should be in the same location. That is to say the position of the

landmark $\vec{P}$ in the kth plus one time-step should not change. The same can be said about the location of the radioactive source and intensity.

$$ {}_i\vec{P}_{k+1} = {}_i\vec{P}_k \qquad \text{Eq. 2.92} $$

$$ {}_i\vec{R}_{k+1} = {}_i\vec{R}_k \qquad \text{Eq. 2.93} $$

A second version of the transition model function as TransitionModel2() which returns identical output with again input more compatible with the numjac() function in order to produce an A matrix. However using the numjac() function is not necessary as it defaults returning something nearly identical to the first transition matrix defined earlier with different dimensions that are compatible with the state vector length present here.

The length of the state vector can change as new landmarks are identified, this will also cause the dimensions of the state transition matrix A to change as well to be compatible. This is an important part of the EKF design. This also means that Q, R and P matrices also require resizing. The functions that handle the resizing will be discussed in the additional helper functions.

2.5.3.2 THE MEASUREMENT MODEL

The measurement model for the vehicle states regarding GPS and gyro measurements again remain untouched mathematically. The same cannot be said with the landmark measurement. When measurements of confirmed landmarks are made, the EKF is told specifically through the state-to-measurement matrix H that they were measured, but that the measurement is made relative to the vehicle. The measurement is done in the vehicle frame, that vector is rotated into the inertial frame to provide a vector from the vehicle to the landmark. For example, if $\vec{P}$ is the location of the landmark and $\vec{z}_p$ is a measurement of the landmark relative to the vehicle position $\vec{x}_{Veh}$. The following equation relates the measurements to the vehicle position in the inertial reference frame.

$$ \vec{z}_p = \vec{P} - \vec{x}_{Veh} \qquad \text{Eq 2.94} $$

Both the landmark position $\vec{P}$ and the vehicle position $\vec{x}_{Veh}$ are part of the state vector $\vec{x}$. The measurement model uses a function $h(x_k)$ in order compute the measurement estimate as shown in equation 2.94. In the program the function that does this is called ObservationModel() and can be seen in appendix A. There is a second version of this function with identical output but different input in order to use the numjac() function to compute the state-measurement-matrix H that is used in the computation of the Kalman Gain matrix.

A measurement of a single landmark can be made within some threshold distance set within the simulation, for most applications this distance is set to 1m. This threshold distance allows for a direct control over the resolution of the environment. The process for handling measurements and landmarks falls primarily within to major functions. The first is within the detect_env3_imu_cleaned() where the general proximity detection process occurs. As positive measurements are made, the state of the vehicle, and the location of the measurement relative to the vehicle are stored in an observation struct. Additionally, the number of new observations made in a single timestep are returned from the detect_env3_imu_cleaned() function. With the observations made, and all relevant information stored, the next step is to process these observations and turn them into useful landmark measurements for the EKF. The function responsible for this is Parse_Observations(). This function takes and receives many inputs and outputs. The purpose of the function is to determine from observations new confirmed landmarks, measured unconfirmed landmarks, upgrade unconfirmed landmarks to confirmed, and determine confirmed landmark measurements. I will better describe this process in the additional helper functions section. For the sake of the measurement model, this function returns a vector of confirmed landmark measurements. This vector is a portion of the measurement vector $\vec{z}_k$.

The next portion of the measurement model is what it predicts for a radiation measurement. Again, the frequency of radiation measurements is one reading per second. The EKF will predict a single radiation measurement based off of the estimated information of the source, and the vehicle position.

Equation 2.95 is used for the measurement process. It says the predicted radiation measurement z_R is a function of the source's intensity I and its distance r away from the vehicle.

$$z_R = \frac{I*\Delta t}{r^2} \qquad \text{Eq. 2.95}$$

$$r = |\vec{P}_{Rad} - \vec{x}_{Veh}| \qquad \text{Eq. 2.96}$$

$$\Delta t = 1\ second \qquad \text{Eq. 2.97}$$

The ObservationModel() uses equations 2.96 and 2.97 in order to compute the single radiation measurement estimate, the estimate only occurs when an actual radiation measurement is present.

With the measurement model being explicitly dependent on the vehicle location it adds a layer of depth to how the Extended Kalman Filter perceives the relationship between the landmark and radiation states with the vehicle states. The design allows the measurements of landmarks and radiation to affect the vehicle position estimates, for both good and bad.

2.5.3.3 THE EXTENDED KALMAN FILTER INITIALIZATION

The same Q, R, and P_0 matrices have to be initialized. The Q matrix is built with estimated errors of the model down the diagonal through the build_Ex() function. The specific values will be stated for each set of results. The R matrix is built with measurement errors using the build_Ez() function, again the specific error values will be states for each data set. The P_0 matrix is initialized with all zeros, at the current dimensions of the state vector.

What is unique about the EKF here is the dynamic size and dimensions of the vectors and matrices. At the start the state vector is initialized with zero landmarks. As observations are made and new landmarks pass the confirmation process, the program inserts the appropriate states and errors into the state vector and matrices respectively. All previous information is untouched.

2.5.3.4 ADDITIONAL HELPER FUNCTIONS

The Parse_Observations() function plays a major role in the observations/measurement process. It is responsible for taking raw observation data and turning it into useful measurements for the EKF to use. In order to explain this function it is important to understand the landmark system. The landmark system has two different types of landmarks, confirmed and unconfirmed. Only confirmed landmarks are estimated by the EKF. The unconfirmed landmarks must meet a minimum threshold strength level in order to upgrade to a confirmed landmark. If unconfirmed landmarks remain unconfirmed for too long, they are removed from the list. The function takes each new raw observation measurement and goes down a checklist in order as follows:

1. Check if measurement was within some threshold distance to a confirmed landmark.
    a. If true, counts as confirmed landmark measurement, jump to step 4.
2. Check if measurement was within some threshold distance to an unconfirmed landmark.
    a. If true, counts as unconfirmed landmark measurement adds to strength, jump to step 4.
3. Create a new unconfirmed landmark to be checked by all following measurements.
4. Check all unconfirmed landmark strength to see if they pass strength threshold.
    a. If true, upgrade unconfirmed landmark to confirmed list.
5. Check all unconfirmed landmark for time elapsed.
    a. If longer than 30 second ago, remove from unconfirmed landmark list.

This function returns the number of confirmed and unconfirmed landmarks and their location. It also returns the strength and time of creation for all unconfirmed landmarks. It returns a portion of the measurement vector corresponding to the landmark measurements, and which landmarks have been measured to be used in the ObservationModel() function.

The next function, fix_state_vector(), uses the information returned regarding confirmed landmarks and checks that with the current state vector. If it finds there are more confirmed landmarks

than are in the current model it increases the length of the state vector and adds in the location of new landmarks. This function is run every timestep, to ensure new landmarks enter the EKF process.

With the size of the state vector changing due to new landmarks, so too must every other matrix used in the EKF process. The Q, and R matrices are reformed every time step to match the the dimensions with the number of states, and measurements by their respective build_Ex() and build_Ez() functions. The function responsible for fixing the Covariance Matrix P is the Fix_Pk() function. Given the current Covariance Matrix P and the corrected state vector, it checks that the dimensions of the Covariance Matrix match the length of the state vector. If it does not match, the rows and columns associated with the new landmarks are added into the old Covariance Matrix, keeping all previous information while adding room for additional covariance updates. This is important because if the dimensions are not fixed the EKF computations are mathematically impossible.

CHAPTER 3: RESULTS

3.1 KF 1 SIMPLE WALL AVOIDANCE

The results shown here are for a simple world with a single wall. The vehicle is initialized at a position of (2m, 5m, 3m) and is told to fly to (9m, 5m, 5m). The results are from the avoidwall_withIMU() function.

Table 3.1 Results 3.1 Measurement Errors

Error Variable for Measurements and R Matrix	Actual Error Value	Value used in R Matrix
σ_{GPS}	1m	1m
$\sigma_{Compass}$	.1rad	.1rad
σ_{IMU}	0 rad	.2rad
σ_{Gyro}	0 rad/s	n/a
σ_{Obj}	n/a	40
σ_{Rad}	n/a	1 count

Table 3.2 Results 3.1 Estimated Model Errors

Error Variable for Q Matrix	Value Used in Q Matrix
σ_{XYZ}	.01m
σ_{V}	.1m/s
$\sigma_{Heading}$	.1rad
σ_{IMU}	.1rad
σ_{Obj}	5
σ_{Rad}	50 counts

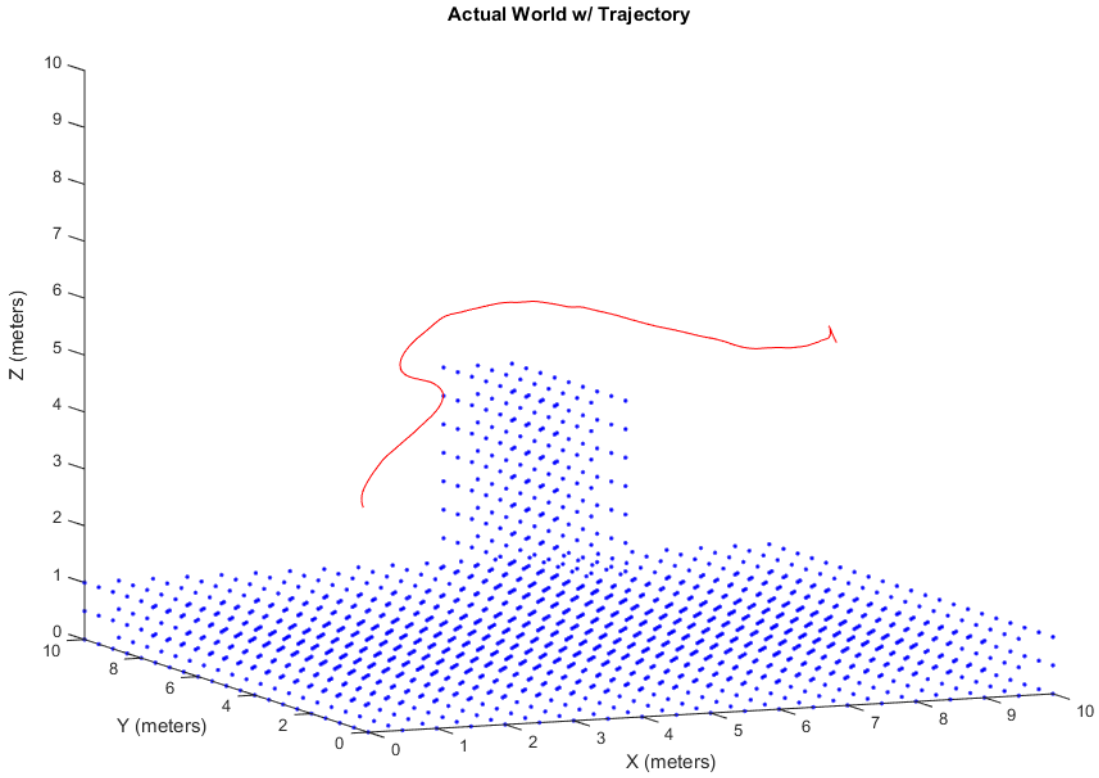

Figure 3.1 Actual World and Trajectory Simple Wall Avoidance: This figure shows the flight path of the vehicle successfully avoiding a wall during flight, simulated using the KF1 model. The vehicle starts on the left and travels to the right (as seen in the image). The blue dots represent the actual world (not estimated) and the red line represents the actual vehicle flight path.

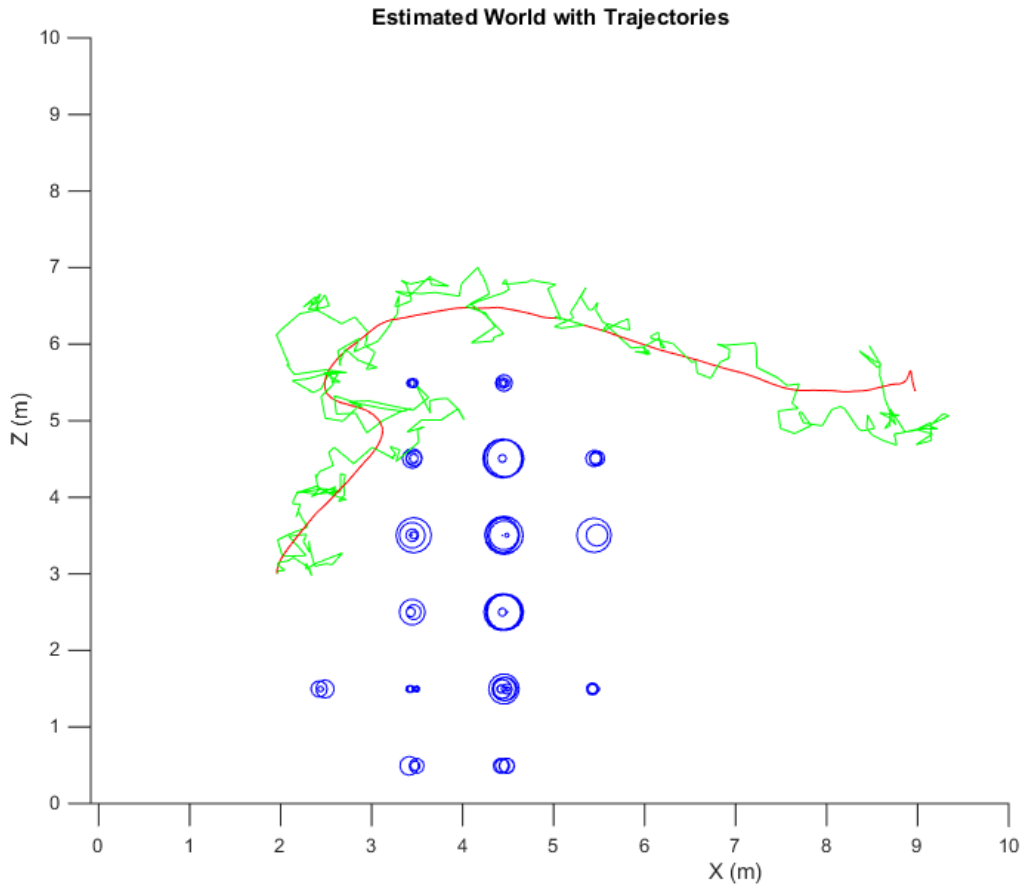

Figure 3.2 Estimated World and Trajectories Simple Wall Avoidance: This figure is a side view down the y-axis of the same flight path. It was simulated using the KF1 model. The blue circles here represent the KF1's estimated world and is what the obstacle avoidance routine sees. The red and green lines represent the actual and estimated flight path respectively.

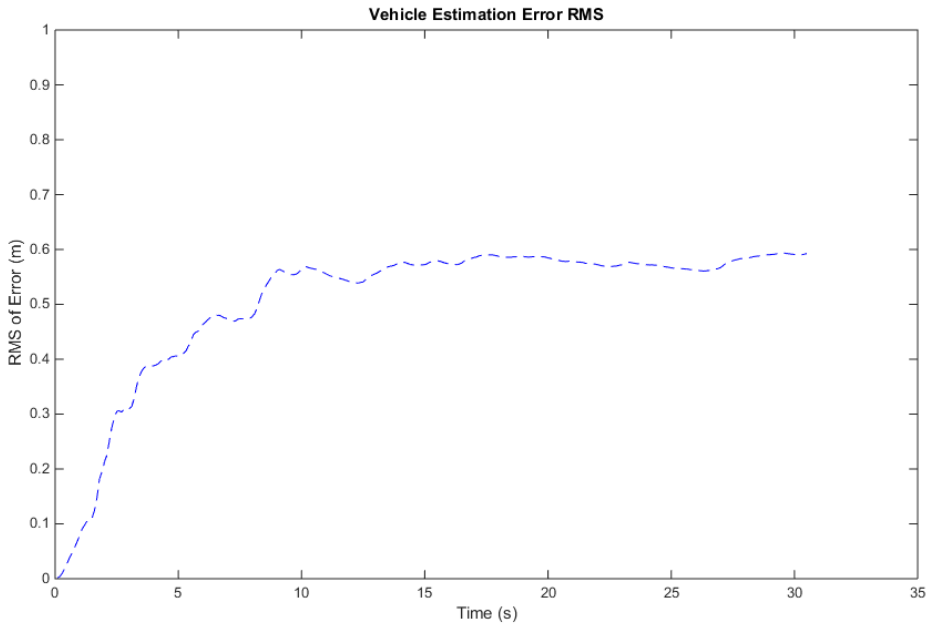

Figure 3.3 Vehicle Estimated Position Error RMS: A graph of the vehicle RMS position error. This is the error in vehicle position estimates from the simulation using KF1 model. The RMS of the error magnitude is calculated at each time step and plotted. The error RMS reaches an asymptote at about .6m after about 12 seconds of flight time.

3.2 KF 1 GRID PATTERN FOR RADIATION MEASUREMENTS

The following set of results is from version 4.2 function highaltgrid_withimu() which can be seen in the full appendix version. It uses the first version of the KF designs, and the vehicle flies in a grid pattern in an attempt to form a useable radiation measurement map to assist in localizing a radioactive source with an activity of 2000 decays per second. The source is located at (1m, 1m, 3m) in the world, which has no obstacles other than a ground surface.

Table 3.3 Results 3.2 Measurement Errors

Error Variable for Measurements and R Matrix	Actual Error Value	Value used in R Matrix
σ_{GPS}	1m	1m
$\sigma_{Compass}$	.1rad	.1rad
σ_{IMU}	0 rad	.2rad
σ_{Gyro}	0 rad/s	n/a
σ_{Obj}	n/a	40
σ_{Rad}	n/a	1 count

Table 3.4 Results 3.2 Estimated Model Errors

Error Variable for Q Matrix	Value Used in Q Matrix
σ_{XYZ}	.01m
σ_{V}	.1m/s
$\sigma_{Heading}$	.1rad
σ_{IMU}	.1rad
σ_{Obj}	5
σ_{Rad}	50 counts

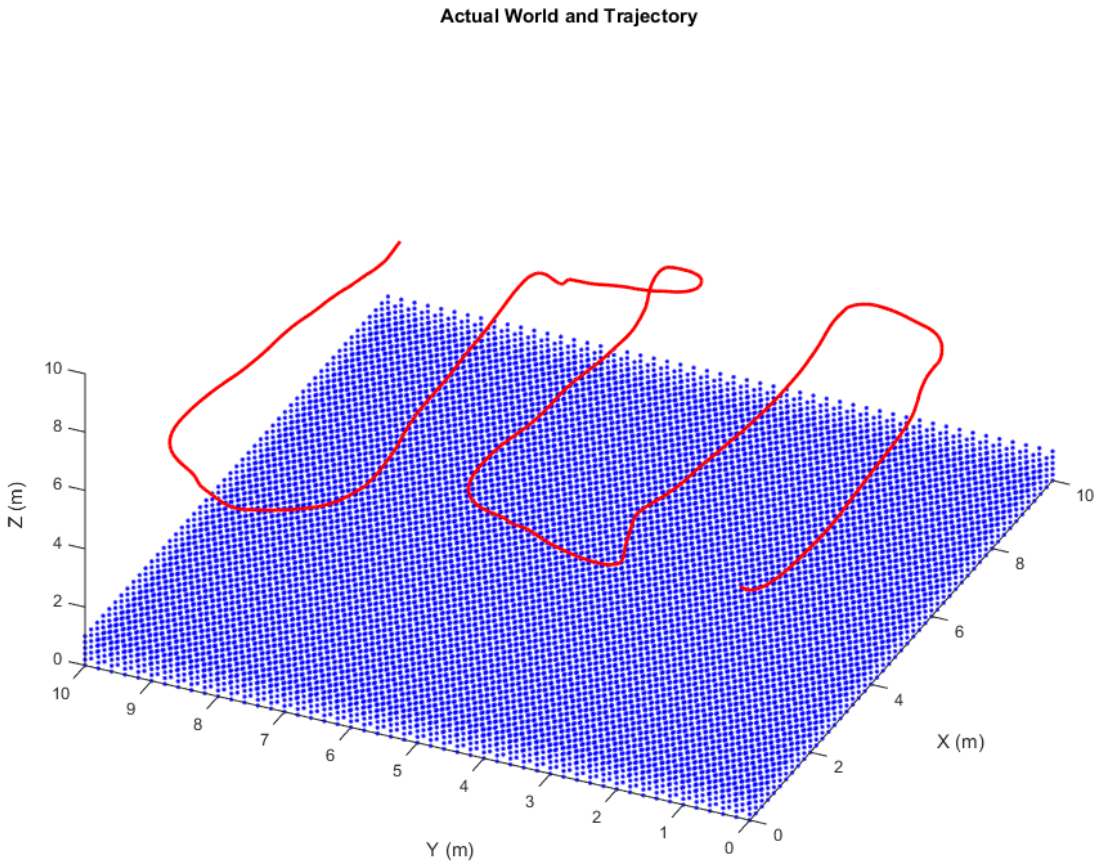

Figure 3.4 Actual World and Grid Pattern Trajectory: These results are the show the vehicle's flightpath (in red) during a grid pattern exercise to explore the radiation estimates. This is done for the KF 1 model. The vehicle is flown high above any detectable obstacles; radiation data is collected for the duration of the flight.

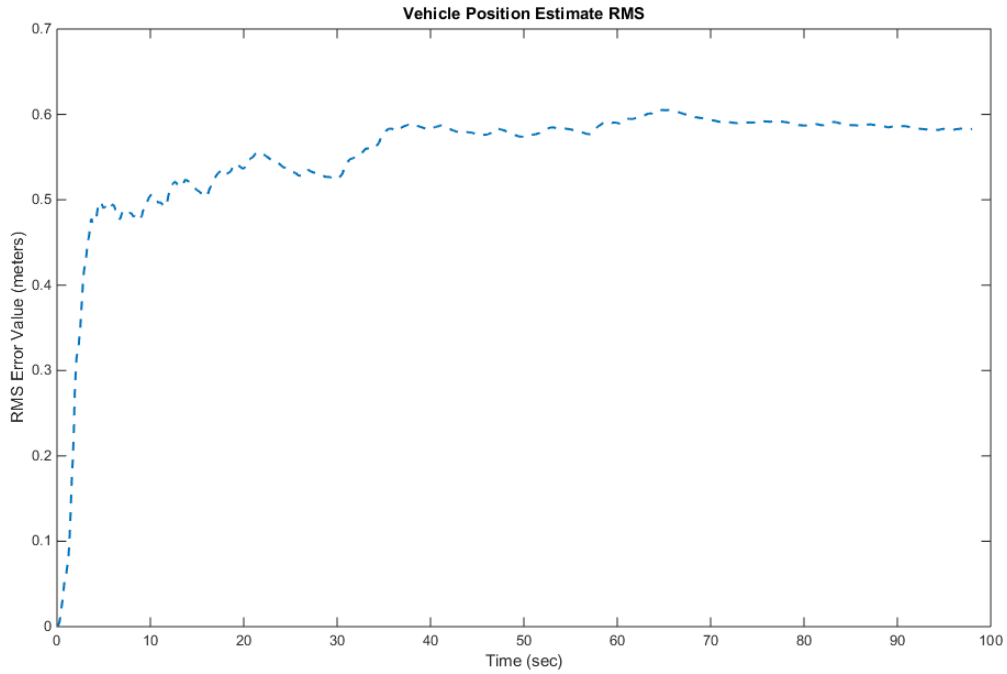

Figure 3.5 Vehicle Estimated Position Error RMS: This figure shows the RMS position error for 3.2 results.

It is done for the estimates in KF1 model. RMS values reach an asymptote of about .6m.

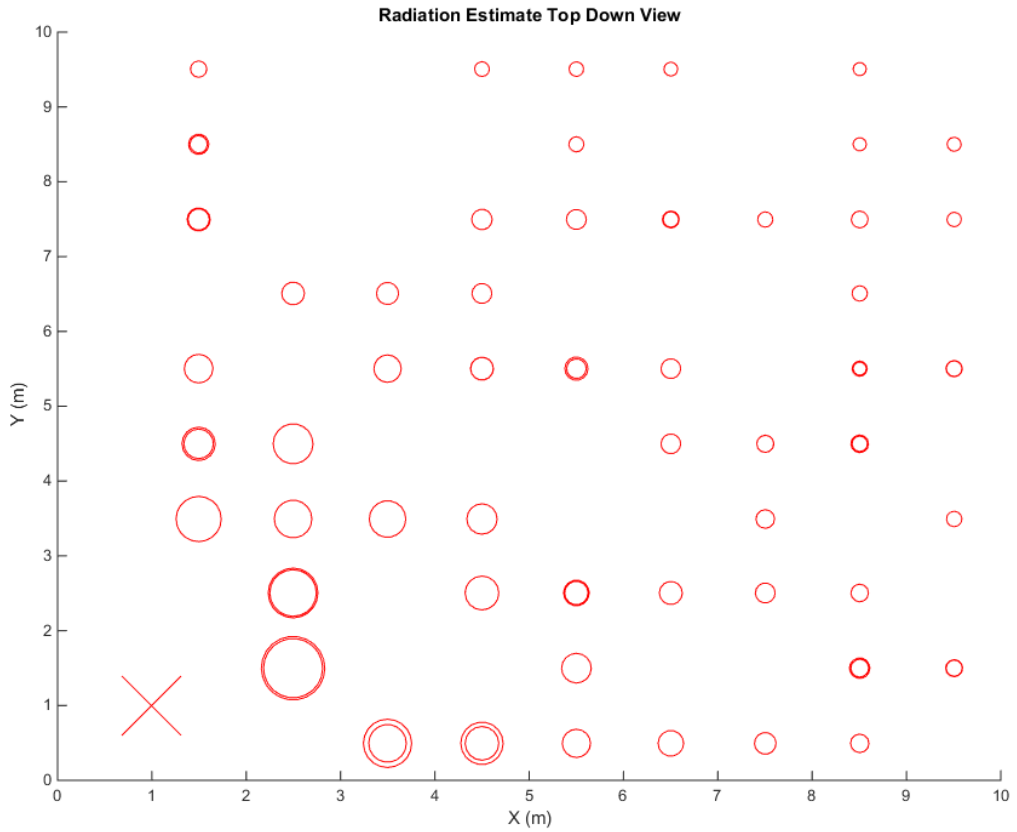

Figure 3.6 Estimated Radiation Map Top Down View: This figure shows the results of the radiation estimates from the grid pattern flight from a top-down view. These estimates are done by the KF1 model. The red circles show where radiation measurements were taken (according to KF position estimates), and the size corresponds to the number of counts estimated. The red "X" shows the location of the radioactive source.

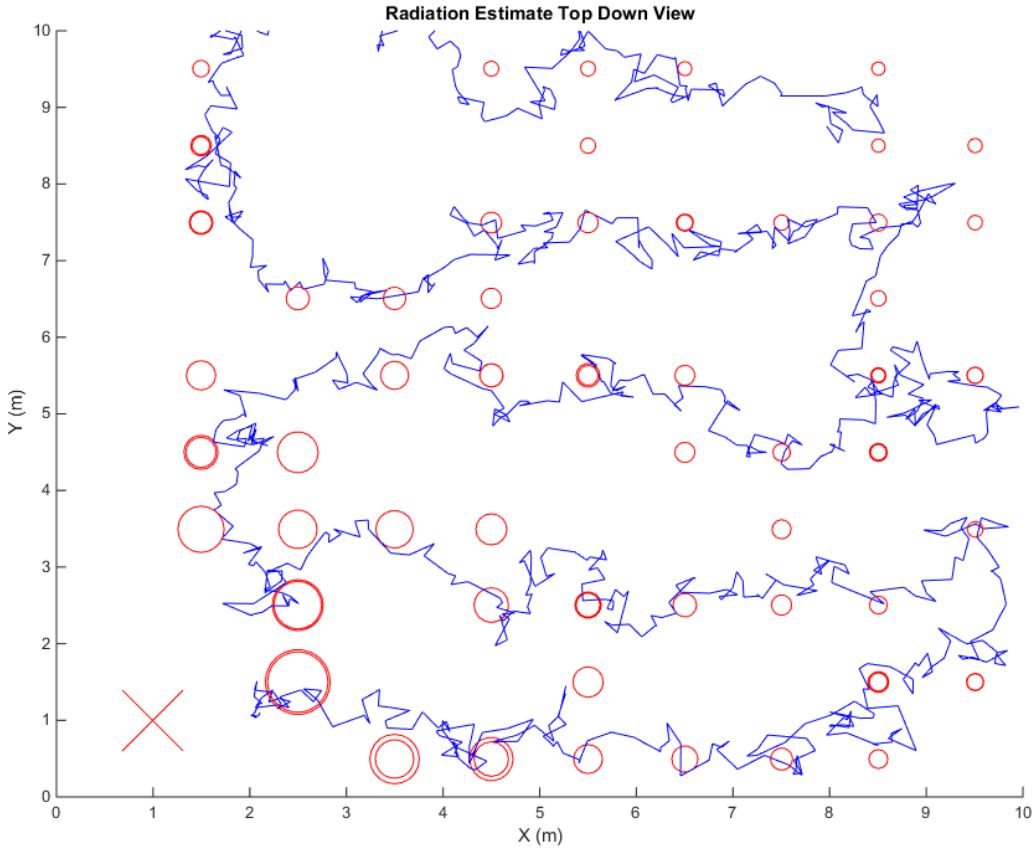

Figure 3.7 Estimated Radiation Map with Estimated Trajectory Top Down View: This figure shows the results of the radiation estimates and vehicle position estimates from a top-down view. These estimates are done by the KF1 model. The red circles show where radiation measurements were taken (according to KF position estimates), and the size corresponds to the number of counts estimated. The red "X" shows the location of the radioactive source. The blue line represents the estimated vehicle flightpath. The estimates flightpath is added to provide context to the location of the red circles, which are based off where the KF believes the vehicle to be when a measurement is made.

3.3 KF 1 ADVANCED WORLD WITH OBSTACLE AVOIDANCE LOW ERROR

The following are the results corresponding to the first version 4.2 of the code that uses the first KF design. The tables 3.6 show the error values used in measurements and matrix construction. The world is a complex example compared to other tested worlds. The vehicle has a single waypoint to navigate to and attempts to avoid obstacles along the way. The vehicle starts at position (5m, 0m, 4m) and is told to navigate to (5m, 9m, 5m). This example is to show the robustness of the first KF object world design and vehicle position estimation. A source with activity 2000 counts per second is placed a location (1m, 1m, 3m).

Table 3.5 Results 3.3 Measurement Errors

Error Variable for Measurements and R Matrix	Actual Error Value	Value used in R Matrix
σ_{GPS}	.5m	1m
$\sigma_{Compass}$	.05rad	.2rad
σ_{IMU}	0 rad	.2rad
σ_{Gyro}	0 rad/s	n/a
σ_{Obj}	n/a	40
σ_{Rad}	n/a	1 count

Table 3.6 Results 3.3 Estimated Model Errors

Error Variable for Q Matrix	Value Used in Q Matrix
σ_{XYZ}	.02m
σ_{V}	.2m/s
$\sigma_{Heading}$	.1rad
σ_{IMU}	.1rad
σ_{Obj}	5
σ_{Rad}	50 counts

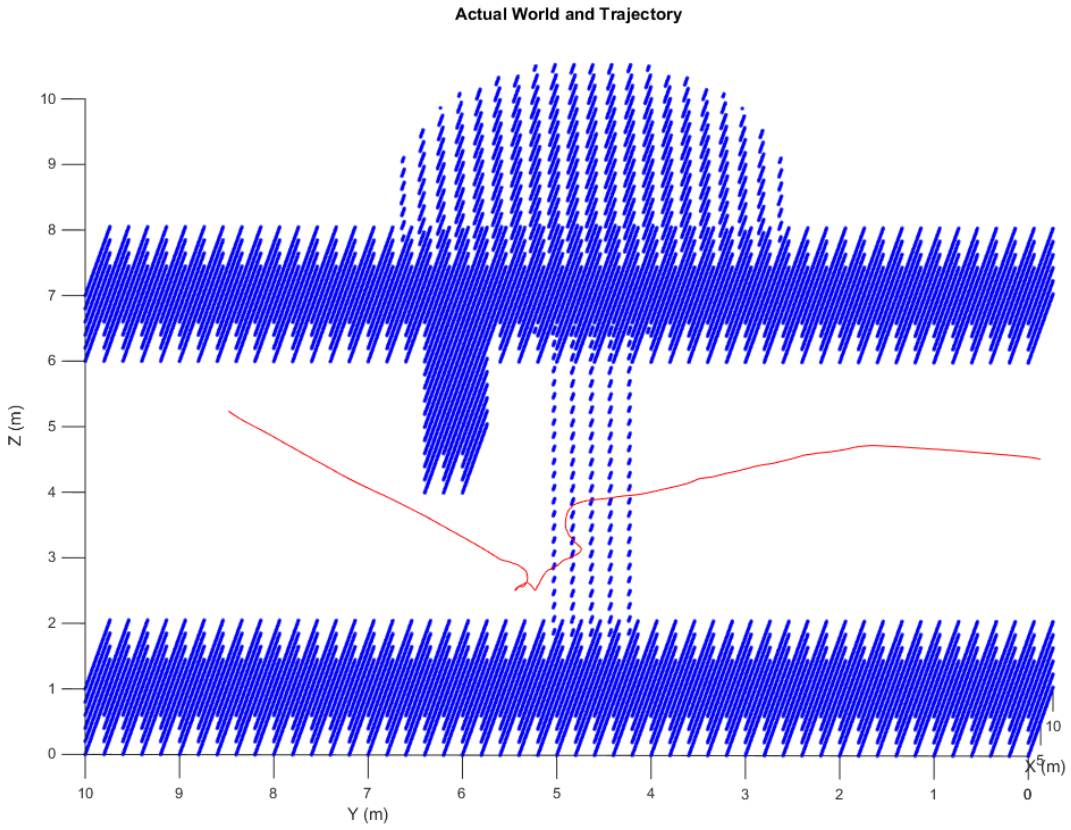

Figure 3.8 Results 3.3 Actual World and Trajectory: This figure shows the vehicle's flightpath in a more complex environment. These results are generated using the KF1 model. The vehicle is told to navigate from right to left as seen. Displayed are the actual vehicle flightpath in red, and the actual world in blue.

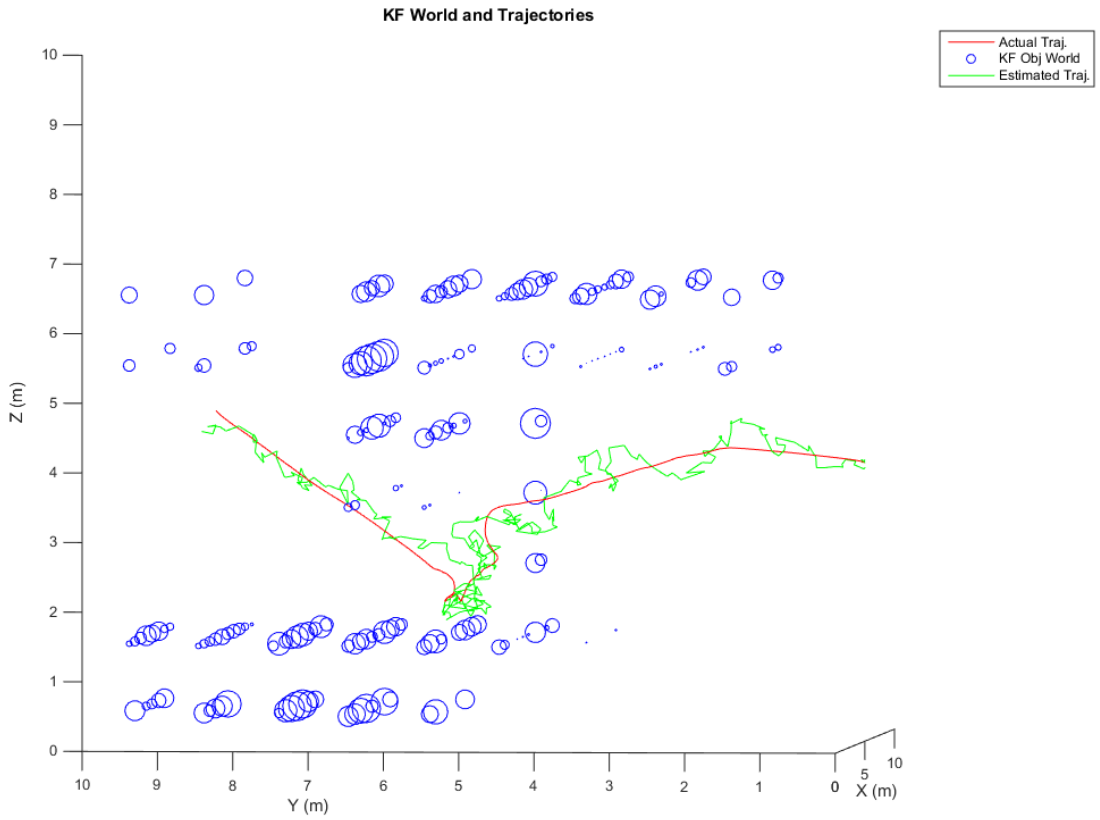

Figure 3.9 Results 3.3 Estimated World with Real and Actual Trajectories: This figure shows the estimated results for the same run as 3.8. These estimates are done by the KF1 model. The red line shows the actual vehicle flight path. The green line represents the estimated flight path. The blue circles correspond to the estimated space. The estimated world looks quite similar to the actual world displayed in figure 3.8.

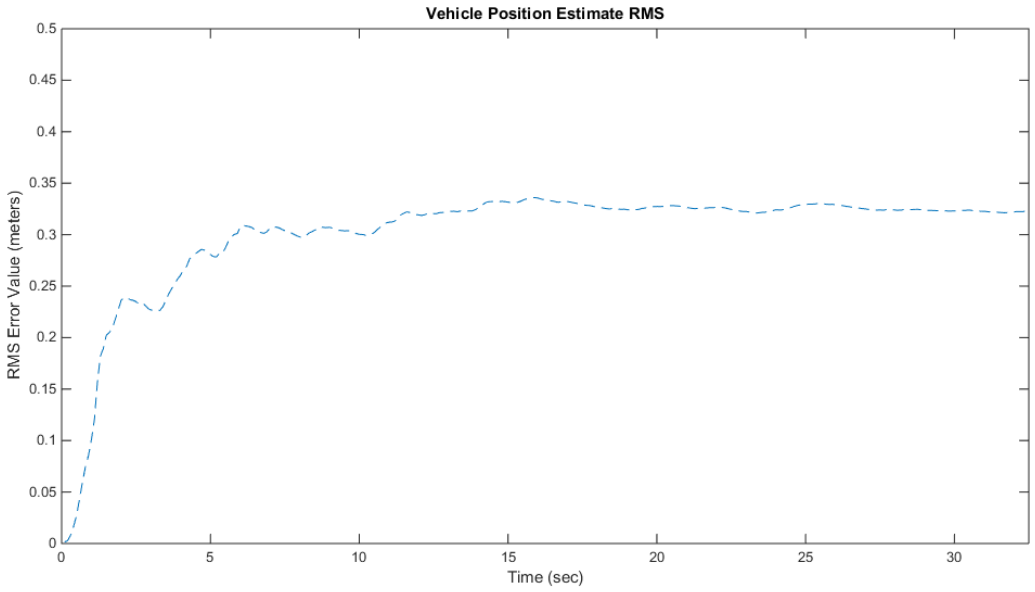

Figure 3.10 Vehicle Estimated Position Error RMS: The figure shows the RMS position error for 3.3 results. The estimates are done by the KF1 model. The RMS settles at around .32m. The values are lower than previous runs due a lower base GPS error.

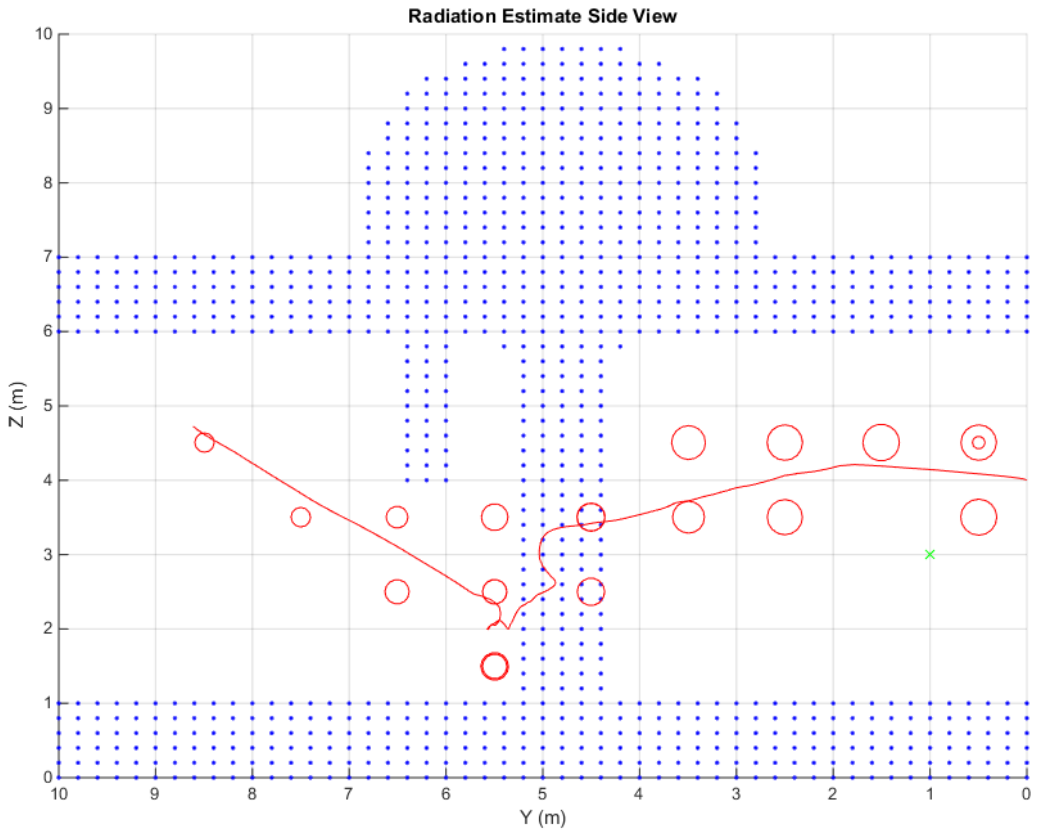

Figure 3.11 Results 3.3 Side View of Radiation Estimates: This figure shows the results of the radiation estimates with vehicle flightpath and world context. These estimates are done by the KF1 model. The red line represents the actual vehicle position. The red circles show the position of the radiation measurement estimates; their size corresponds to the number of counts estimated at that location. The source is located at the green "x". The actual world is shown by the blue dots.

3.4 EKF 2 SIMPLE WORLD LOITER

In this experiment the vehicles ability to detect the environment is looked at. The vehicle holds position (10m, 10m, 2m) and faces 45 degrees from the x-axis. The vehicle loiters for 30 seconds of total flight time, detecting the radiation in the environment every second.

Table 3.7 Results 3.4 Measurement Errors

Error Variable for Measurements and R Matrix	Actual Error Value	Value used in R Matrix
σ_{GPS}	1m	1m
$\sigma_{Compass}$	.1rad	.1rad
σ_{IMU}	.1 rad	.1 rad
σ_{Gyro}	0 rad/s	n/a
σ_{Obj}	n/a	.001
σ_{Rad}	n/a	1 count

Table 3.8 Results 3.4 Estimated Model Errors

Error Variable for Q Matrix	Value Used in Q Matrix
σ_{XYZ}	.02m
σ_{V}	.2m/s
$\sigma_{Heading}$	.1rad
σ_{IMU}	.1rad
σ_{Obj}	.5
σ_{Rad}	50 counts

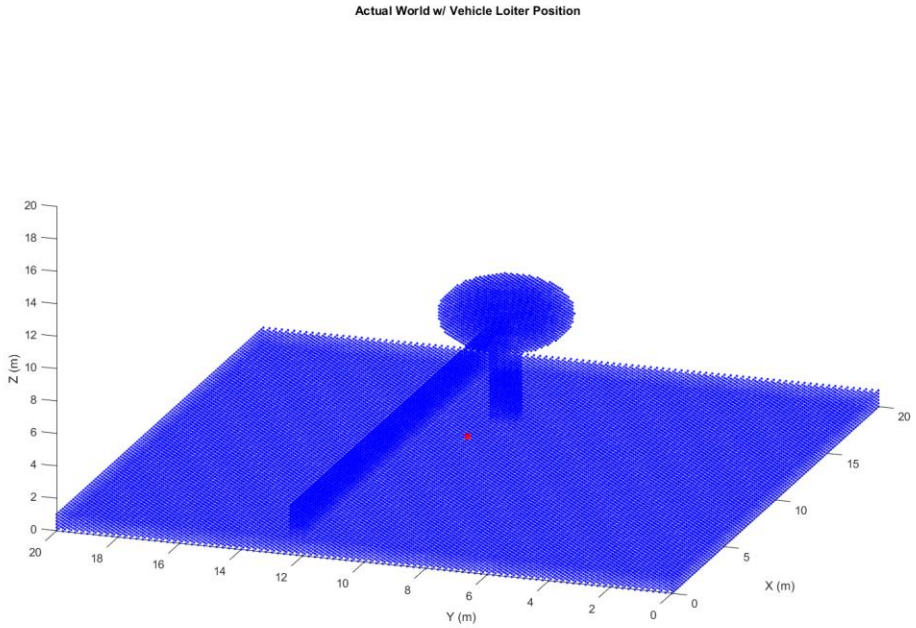

Figure 3.12 Actual World w/ Vehicle Loiter Position: This figure shows the world and the position of the vehicles loiter. The simulation is done for the EKF 2 model. The vehicle does not move during the simulation, with the exception of some perturbation from the loiter position due to autopilot controls. The actual world is represented by the blue dots, the red "x" represents the vehicle's loiter position.

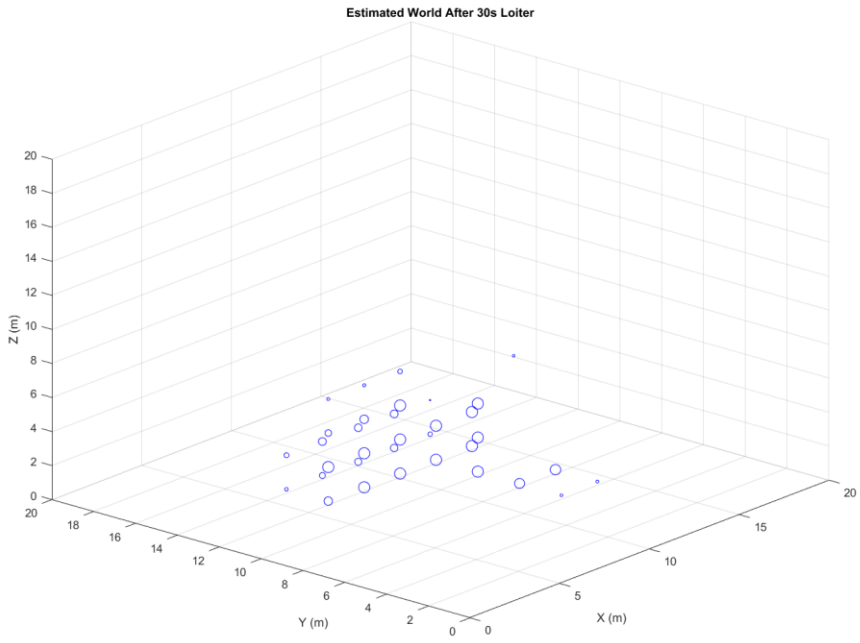

Figure 3.13 Estimated World After 30s Loiter: This figure shows the results of the estimated world after 30 seconds of flight time. For the duration of the loiter the vehicle is facing 45 degrees above the x-axis. The estimates are done by the EKF 2 model. The estimates are copied and pasted to a larger memory world based off the estimated vehicle position from the EKF 2 model. The blue circles represent occupancy estimates at their locations.

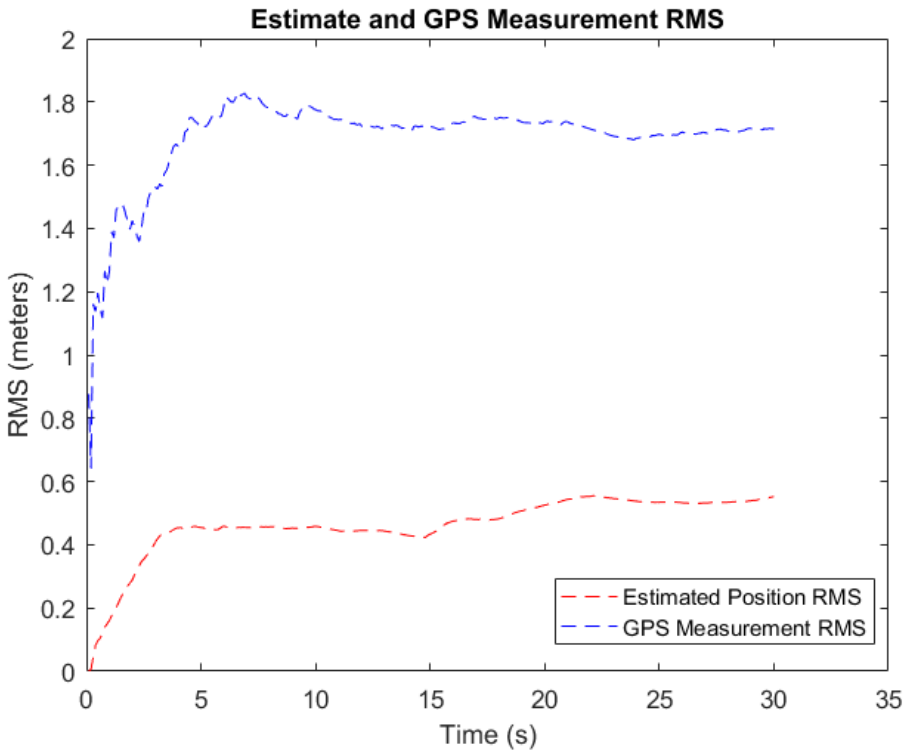

Figure 3.14 RMS Position Error for Estimates and GPS Measurements: This figure shows the RMS position error for both the GPS measurements and the vehicle position estimates. The estimates are done by the EKF 2 model and the base GPS 1-D error is 1m. The dashed red line is the RMS position error in the estimate, the blue is the RMS position error in the raw GPS measurements.

3.5 EKF 3 SIMPLE WORLD OUTDOOR

The following output is for version 7 of simulation which uses the third version of the EKF. It flies around a simple outdoor environment with a fence and treelike object. The purpose of these results is to illustrate the EKF in the context of an outdoor environment. The vehicle starts at (5m, 9m, 2m) and navigates 5 different waypoints. The radioactive source is located at (5m, 12m, 5m) with an activity of 2000 decays per second. The initial guess of the location of the source is (10m, 10m, 2m).

Table 3.9 Results 3.5 Measurement Errors

Error Variable for Measurements and R Matrix	Actual Error Value	Value used in R Matrix
σ_{GPS}	.75m	.75m
$\sigma_{Compass}$	.1rad	.1rad
σ_{IMU}	.1 rad	.1rad
σ_{Gyro}	.1 rad/s	n/a
$\sigma_{Landmarks}$	n/a	1m
σ_{Rad}	n/a	5 counts

Table 3.10 Results 3.5 Estimated Model Errors

Error Variable for Q Matrix	Value Used in Q Matrix
σ_{XYZ}	.02m
σ_{V}	.1m/s
$\sigma_{Heading}$	.1rad
σ_{IMU}	.1rad
$\sigma_{Landmark}$	.5m
σ_{RadPos}	.25m
σ_{RadInt}	100 counts/sec

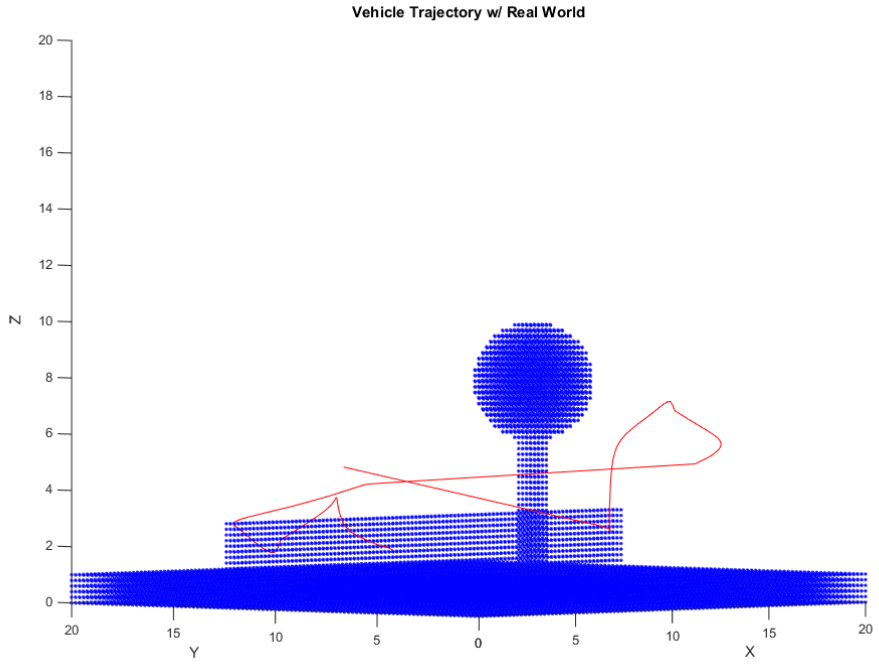

Figure 3.15 Results 3.5 Vehicle Trajectory with Real World: This figure shows both the actual UAV flightpath and actual world. This flight is done during EKF 3 model in an outdoor environment with a simple obstacle avoidance routine. The red line shows the actual UAV flightpath, the blue dots represent the actual world.

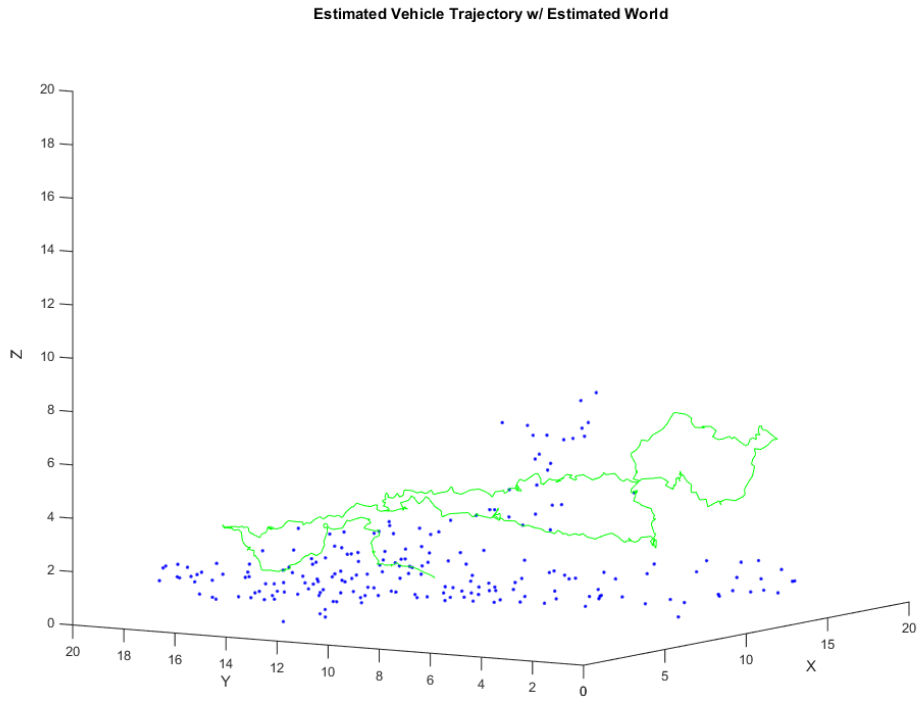

Figure 3.16 Results 3.5 Estimated Trajectory and World: This figure shows the estimated vehicle flightpath and estimated world for the run shown above in figure 3.15. These estimates are done by the EKF 3 model. The green line shows the estimated vehicle flight path. The blue dots correspond to the estimated confirmed landmark locations.

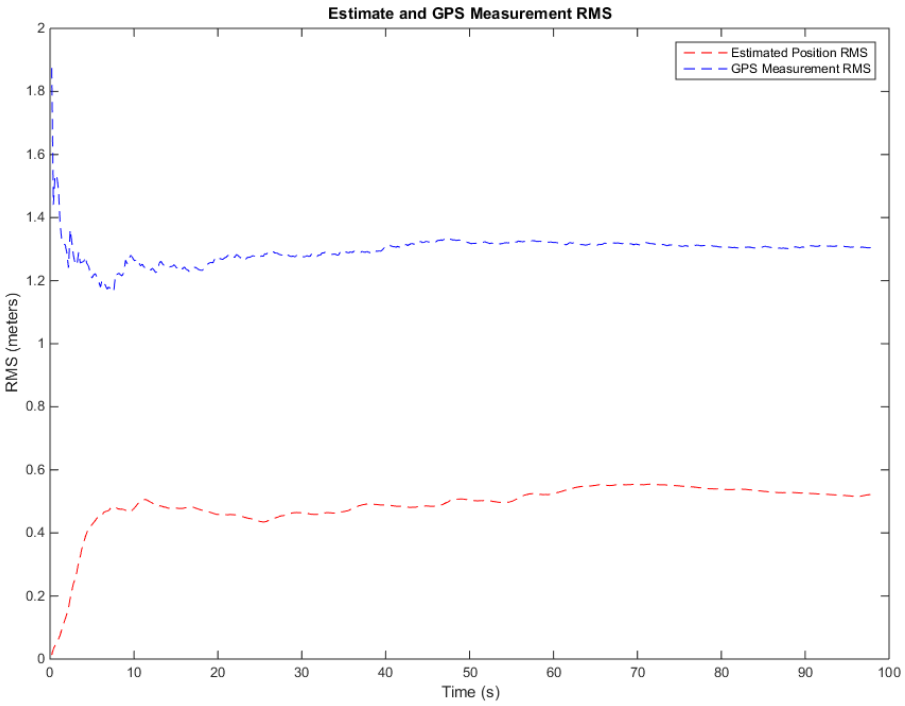

Figure 3.17 Results 3.5 RMS Position Error for Both Estimates and GPS Measurements: This figure shows the RMS position error for both estimates and raw GPS measurements. The estimates are done by the EKF 3 model. The dashed red line represents the RMS position error for the estimates. The dashed blue line represents the RMS position error for the GPS measurements.

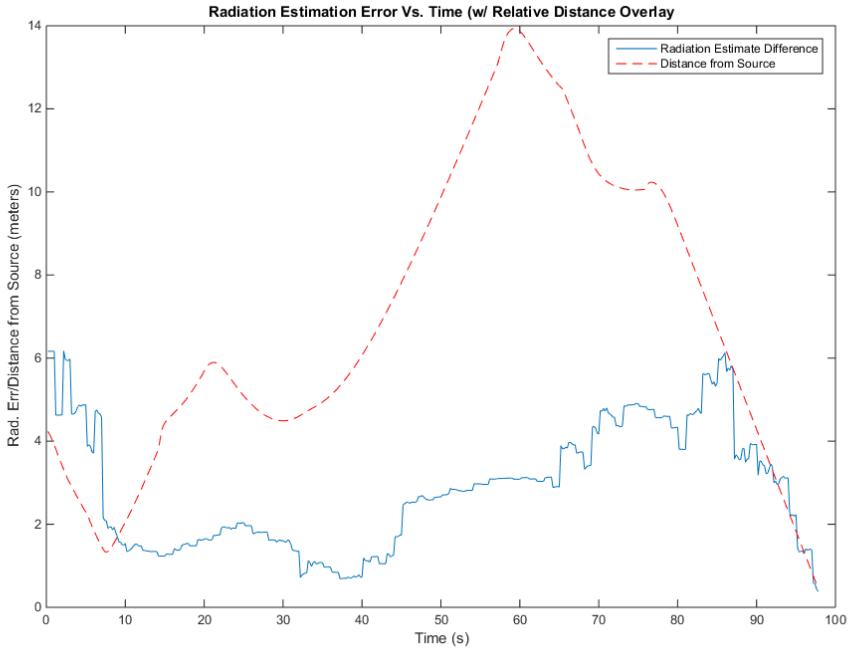

Figure 3.18 Results 3.5 Radiation Estimate Error w/ Distance Overlay: The figure shows the error in the radioactive source location estimates, the distance from the source is also displayed to provide context to the estimates. The estimates are done by the EKF 3 model. The blue line shows the error in the radiation location estimates. The dashed red line shows the distance the vehicle is away from the source at that time. There is a correlation between accuracy and distance to radioactive source.

3.6 EKF 3 URBAN

The following results are from version 7, using the 3rd EKF design. The function motion_withIMU_EKF_urban() flies the drone through an environment that mimics and urban setting. The run has an increased compass error to illustrate performance under an increased error environment. The purpose of these results is to display general results in the urban setting.

Table 3.11 Results 3.6 Measurement Errors

Error Variable for Measurements and R Matrix	Actual Error Value	Value used in R Matrix
σ_{GPS}	1m	1m
$\sigma_{Compass}$	.25rad	.25rad
σ_{IMU}	.1 rad	.1rad
σ_{Gyro}	.1 rad/s	n/a
$\sigma_{Landmarks}$	n/a	1m
σ_{Rad}	n/a	5 counts

Table 3.12 Results 3.6 Estimated Model Errors

Error Variable for Q Matrix	Value Used in Q Matrix
σ_{XYZ}	.02m
σ_{V}	.1m/s
$\sigma_{Heading}$	.02rad
σ_{IMU}	.1rad
$\sigma_{Landmark}$	.5m
σ_{RadPos}	.25m
σ_{RadInt}	100 counts/sec

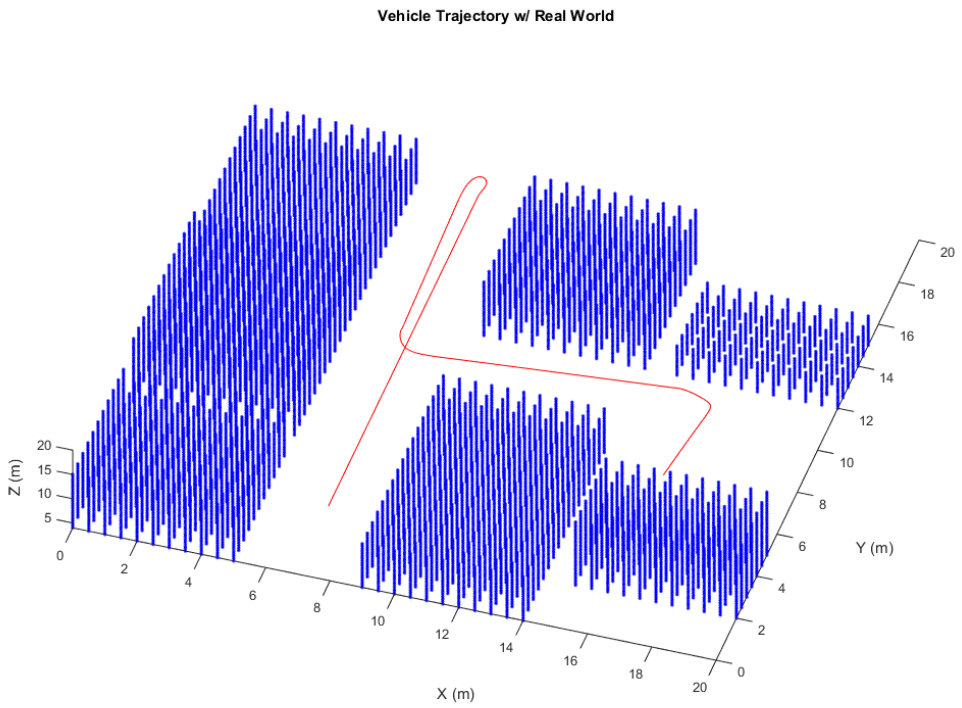

Figure 3.19 Results 3.6 Actual World and Trajectory: The figure displays the actual vehicle flightpath and world for an urban environment. The vehicle is flown during the implementation of the EKF 3 model without obstacle avoidance. The vehicle's flightpath is represented by the solid red line and the world is represented by the blue dots. Here the scale of the Z-axis goes from 5m to 20m, by doing this it removes the ground so that the buildings can be resolved from the surface.

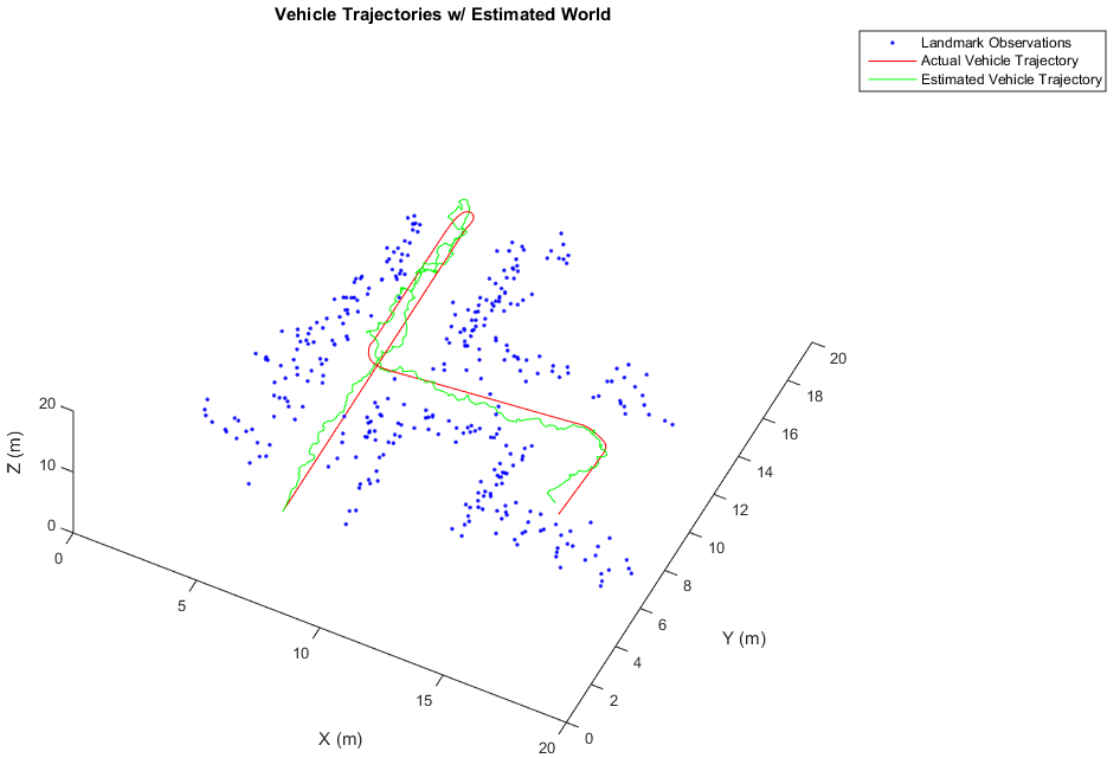

Figure 3.20 Results 3.6 Estimated World and Trajectory: This figure shows the estimated results with the actual flightpath for reference. All estimates were made by the EKF 3 model. The red line represents the actual vehicle flightpath, the blue dots show the estimated landmark positions, the green line represents the estimates vehicle flightpath.

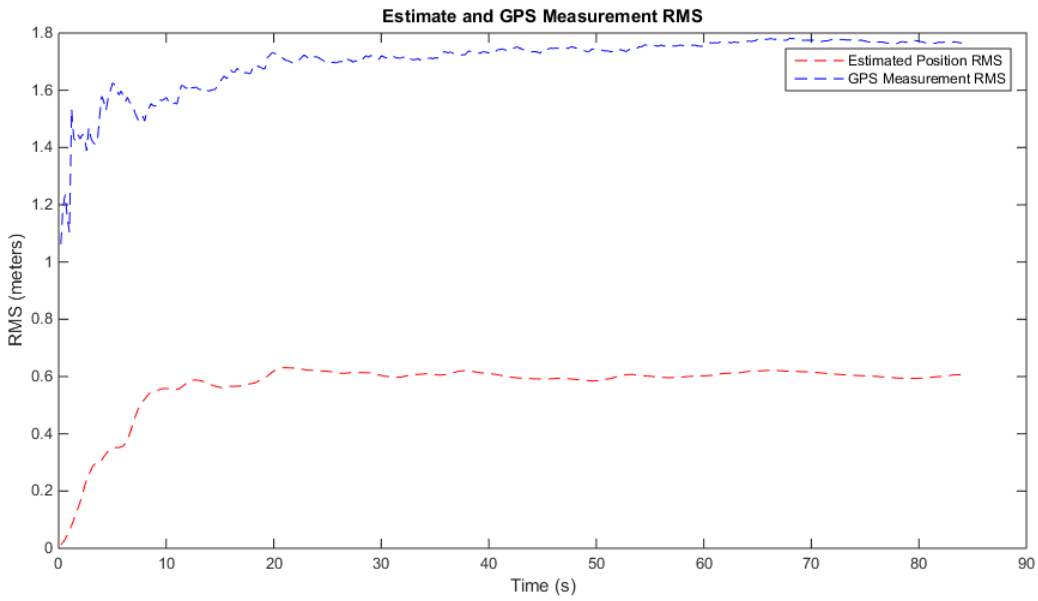

Figure 3.21 Results 3.6 RMS Position Error for Estimates and GPS Measurements: This figure shows the RMS position error for both the estimates and GPS measurements. The RMS is computed at each time step for both the estimated and measured position values. The dashed red line shows the RMS position error for the estimates and the dashed blue line show the RMS position error for the GPS measurements.

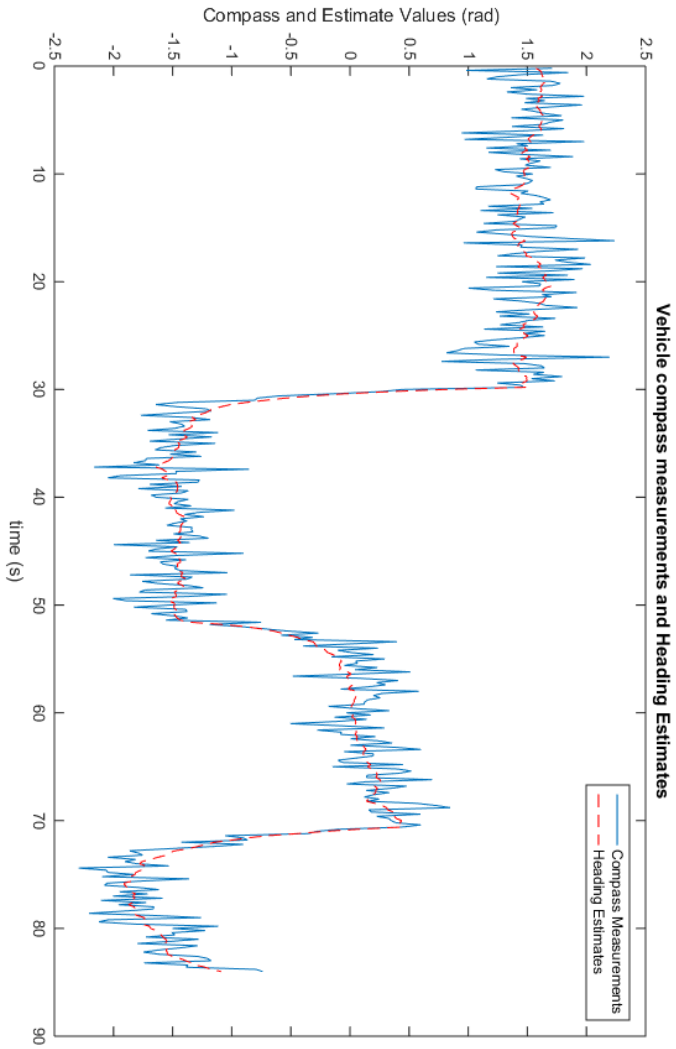

Figure 3.22 Results 3.6 Vehicle Compass Measurements and Heading Estimates: This figure shows the estimated heading and compass measurements of the vehicle. The estimates are done by the EKF 3 model. This solid blue line shows the compass measurements and the dashed red line show the estimated heading. This figure is present to show the change in vehicle heading as the vehicle conducts its flight, and serves as a visual to the performance of other estimated states of the EKF 3 model.

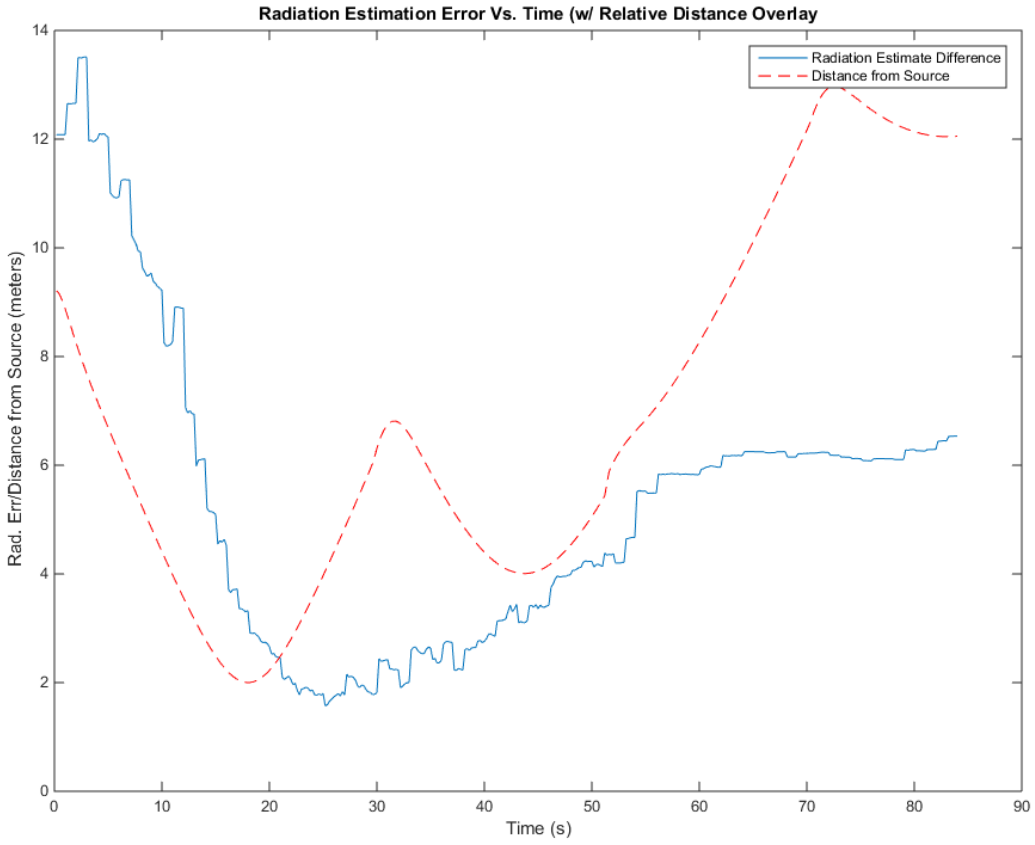

Figure 3.23 Results 3.6 Radiation Estimate Error w/ Distance Overlay: This figure displays the error in the estimated location of the radioactive source and the vehicle's distance from the source for context. The estimates are generated by the EKF 3 model; the distance is the actual distance rather than the estimated distance. The blue line represents the error in the localization estimate, the dashed red line shows the actual vehicle distance from the source.

3.7 EKF 3 WALL AVOIDANCE VERSUS GPS MEASUREMENT

The following results are from several runs of the motion_SimpleWallAvoidance() function seen in appendix A. Each run varies the GPS measurement error. The purpose of this experiment is to understand the performance of the EKF in the context of declining position measurements. Seven different GPS error values were used. Tables 3.13 and 3.14 displays the values used in the experiment. 20 runs were done for each GPS measurement error setting to determine a rough percentage of failure.

Table 3.13 Results 3.7 Measurement Errors

Error Variable for Measurements and R Matrix	Actual Error Value	Value used in R Matrix
σ_{GPS}	.5,1,1.5,2,3,4,5 (m)	.5,1,1.5,2,3,4,5 (m)
$\sigma_{Compass}$	.1rad	.1rad
σ_{IMU}	.1 rad	.1rad
σ_{Gyro}	.1 rad/s	n/a
$\sigma_{Landmarks}$	n/a	1m
σ_{Rad}	n/a	5 counts

Table 3.14 Results 3.7 Estimated Model Errors

Error Variable for Q Matrix	Value Used in Q Matrix
σ_{XYZ}	.02m
σ_{V}	.1m/s
$\sigma_{Heading}$	.1rad
σ_{IMU}	.1rad
$\sigma_{Landmark}$	.5m
σ_{RadPos}	.25m
σ_{RadInt}	100 counts/sec

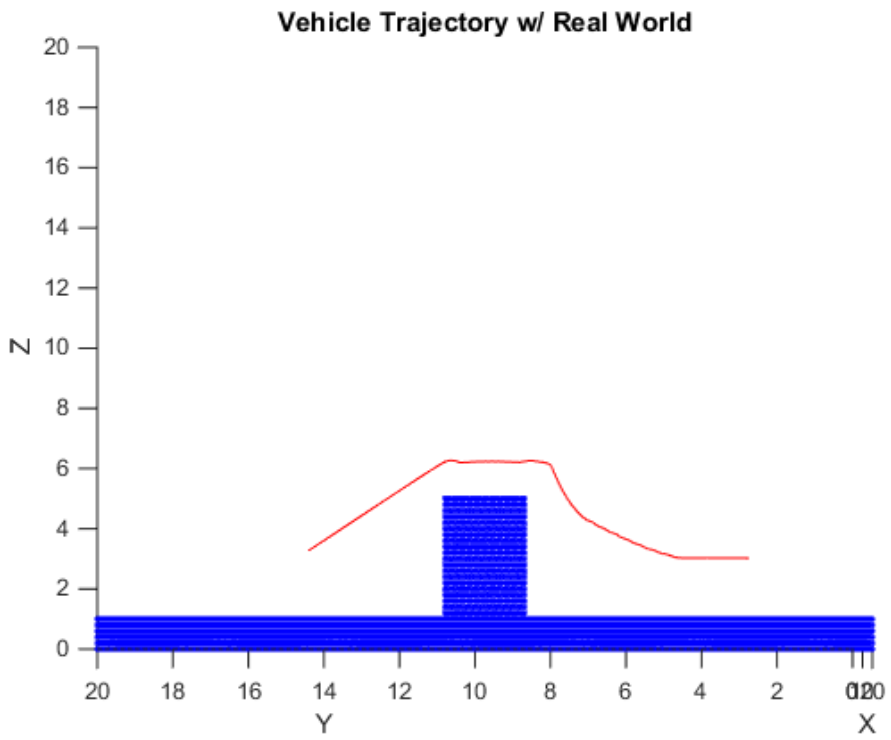

Figure 3.24 Results 3.7 Actual World and Trajectory Success Example: This figure shows the actual flight patch and actual world for a successful obstacle avoidance run. This run us done using the EKF 3 model and simple obstacle avoidance routines. The red like represents the actual vehicle flightpath, the blue dots represent the actual world.

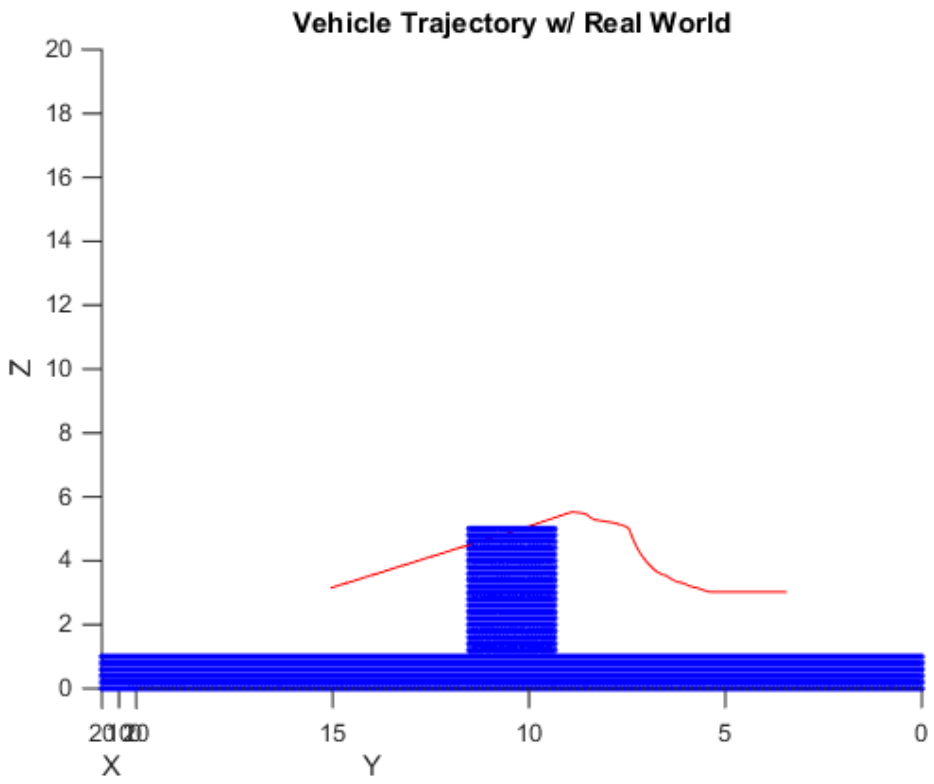

Figure 3.25 Results 3.7 Actual World and Trajectory Failure Example: This figure shows the actual flightpath and world for an unsuccessful obstacle avoidance run. The EKF 3 is used to estimate vehicle position and world. Results from the EKF 3 model are used in simple obstacle avoidance. The red line shows the actual vehicle flight path, the blue dots show the actual world.

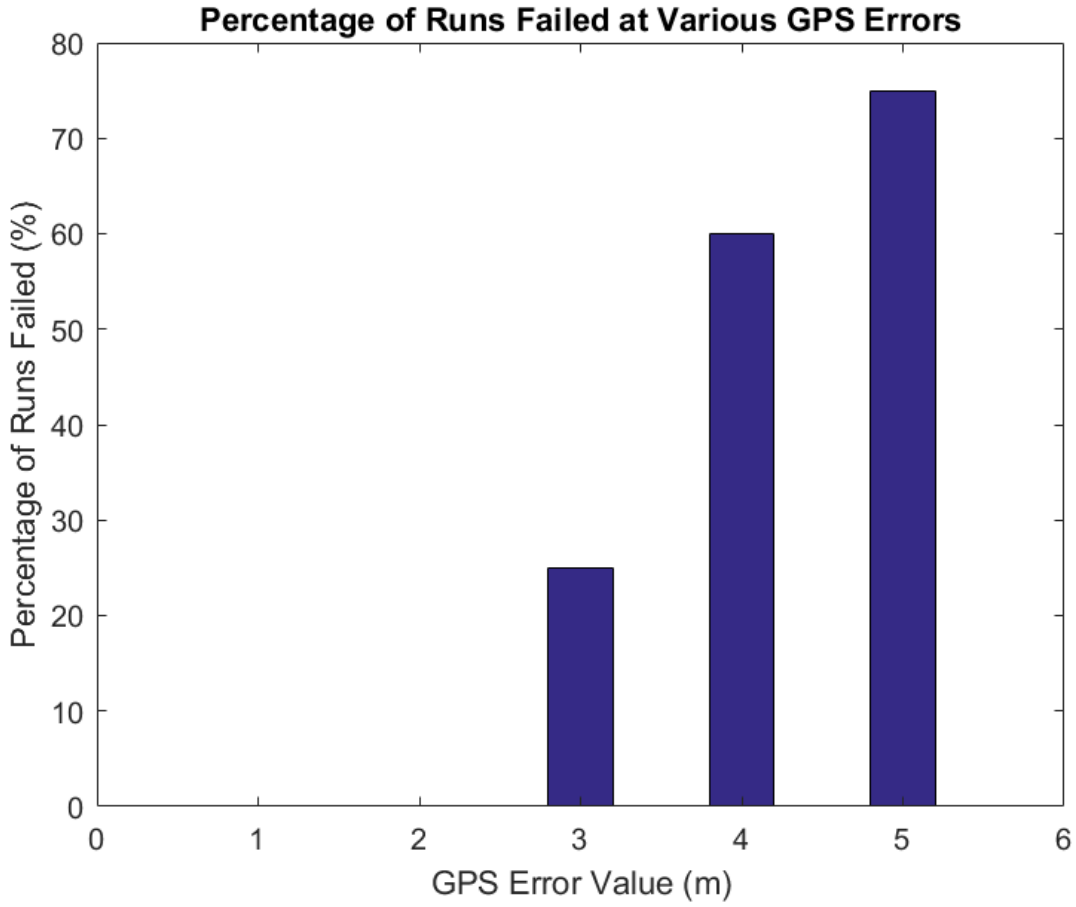

Figure 3.26 Results 3.7 Percentage of Runs Failed at Various GPS Errors: This figure shows the percentage of runs fails at various GPS error values. GPS error values tested were .5,1,1.5,2,3,4,5 (m). Each GPS error was run 20 times with the EKF 3 model estimates. The actual vehicle position was checked to see if it falls within the boundaries of the wall, and returned a fail if true. Simple wall avoidance shows promise up to 2m of 1-D GPS error.

3.8 EKF 3 TESTING THE RELATIONSHIP BETWEEN VEHICLE POSITION AND RADIATION

In this set of experiments, we are looking to see if better radiation estimates and measurements lead to better vehicle localization. In the measurement model, the correlation between the vehicle position, radiation state variables, and the measurement can be seen above in equations 2.95 and 2.96. This model produces non-zero components for both the state-to-measurement matrix and the covariance matrix. Again, we wanted to test the effect of radiation measurements on how it affects the position estimates of the EKF. To do this, the vehicle flies nearby a source with activity 4000 counts per second. At its closest point, it is 1 meter away. The vehicle begins at the position (10m, 6m, 7m) and flies to waypoint (10m, 12m, 7m) then flies back. Two components are changed throughout these results: the GPS error, and the initial guess position of the source(the estimated error in the sources location also to set a confidence level in the source location estimates).

Each set of figures corresponds to different GPS errors, starting at 1m, increasing to 3m. Each figure has three different curves, each representing a different initial guess of the source position. Each parameter was run 5 times. Displayed is the average over 5 runs in order to look for a general trend in higher vehicle position estimation accuracy. Tables 3.15 and 3.16 shows the values used for the experiment minus the variables that change.

Table 3.15 Results 3.8 Measurement Errors

Error Variable for Measurements and R Matrix	Actual Error Value	Value used in R Matrix
σ_{GPS}	varies	varies
$\sigma_{Compass}$	.1rad	.1rad
σ_{IMU}	.1 rad	.1rad
σ_{Gyro}	.1 rad/s	n/a
$\sigma_{Landmarks}$	n/a	1m
σ_{Rad}	n/a	4000 counts/s

Table 3.16 Results 3.8 Estimated Model Errors

Error Variable for Q Matrix	Value Used in Q Matrix
σ_{XYZ}	.02m
σ_{V}	.1m/s
$\sigma_{Heading}$	.1rad
σ_{IMU}	.1rad
$\sigma_{Landmark}$	.5m
σ_{RadPos}	varies
σ_{RadInt}	1 counts/sec

Table 3.17 Results 3.8 Table of Tested Values 1m GPS Error

σ_{GPS}	1m
σ_{RadPos}	R:1m B:5m G:3m

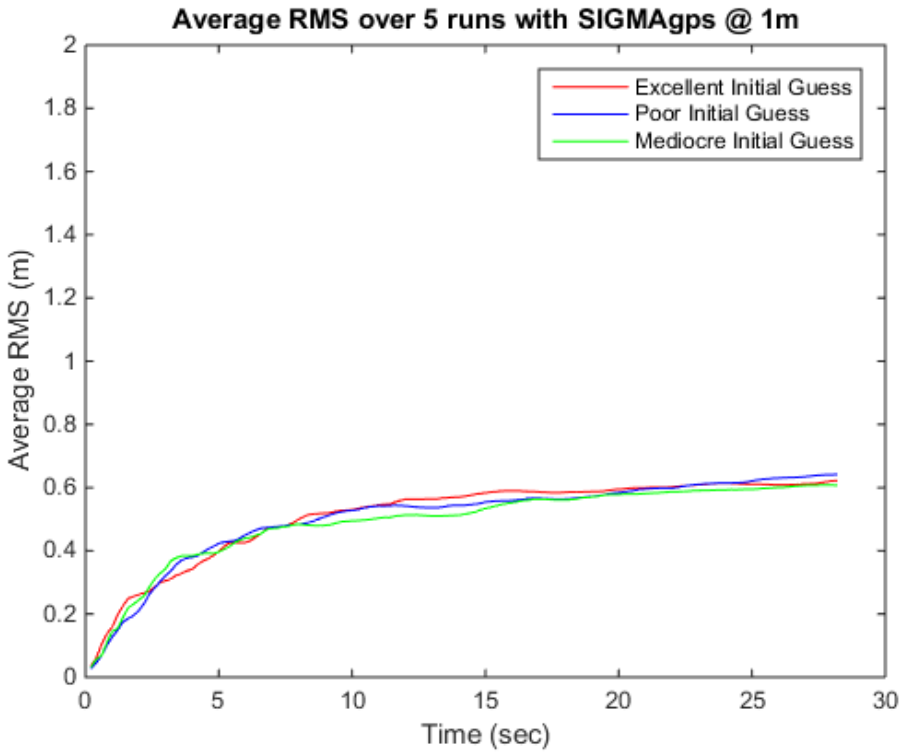

Figure 3.27 Results 3.8 RMS Position Error over 5 Runs 1m GPS error: This figure displays the RMS position error for three runs with varying initial guesses in source radioactive source positions with a GPS error of 1m. The estimates are done by the EKF 3 model. Each initial position guess was run 5 times, the average RMS position error for the 5 runs is computed and plotted for each time step. The RMS position error for the excellent, poor, and mediocre initial guess are the red, blue, and green lines respectively.

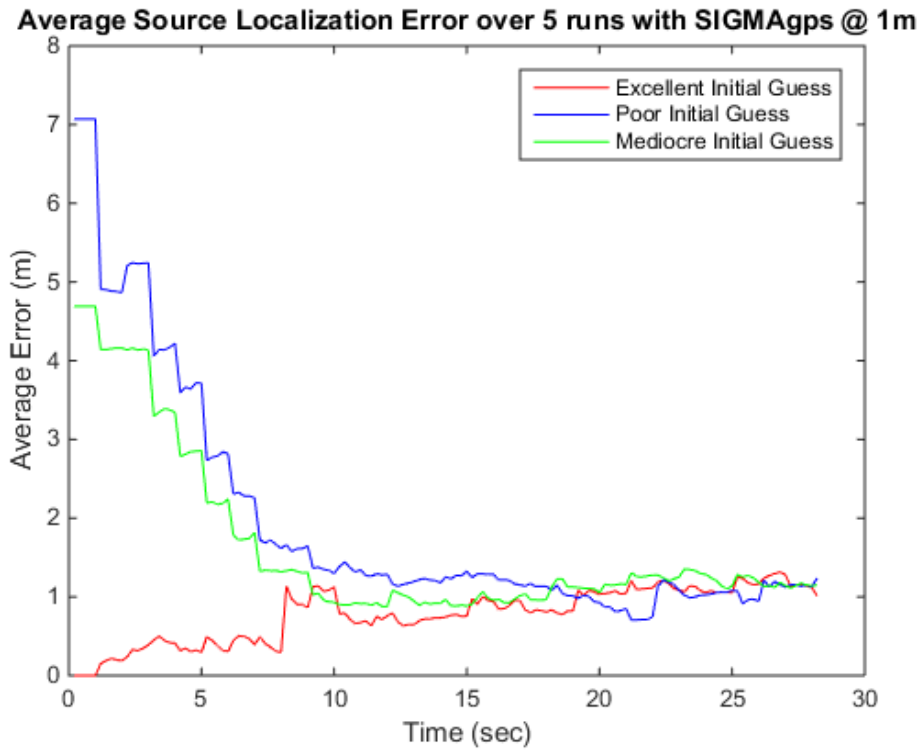

Figure 3.28 Results 3.8 Average Source Localization Error over 5 Runs 1m GPS Error: This figure shows the error in the source localization estimates with varying initial guesses with a GPS error of 1m. The estimates are done by the EKF 3 model. The simulation was run 5 times for each initial position guess. The average error at each time step is computed and plotted. The localization estimate error for the excellent, poor, and mediocre initial guess are the red, blue, and green lines respectively.

Table 3.18 Results 3.8 Table of Tested Values 2m GPS Error

σ_{GPS}	2m
σ_{RadPos}	R:1m B:5m G:3m

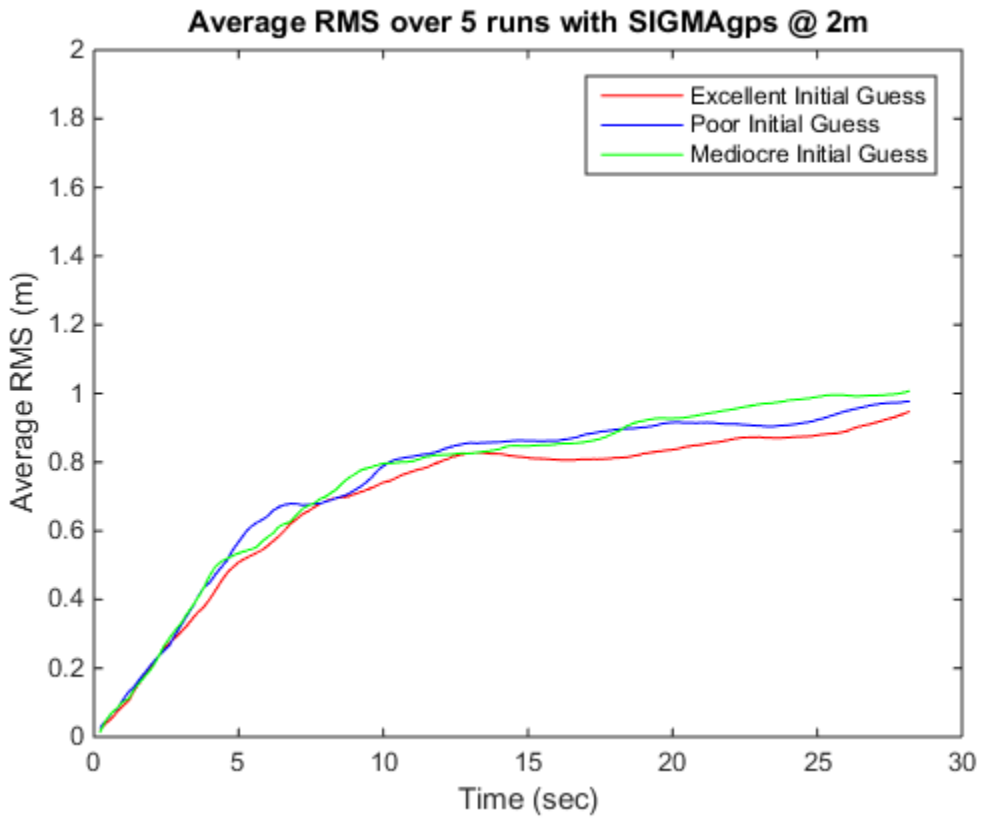

Figure 3.29 Results 3.8 Average RMS over 5 Runs 2m GPS error: This figure displays the RMS position error for three runs with varying initial guesses in source radioactive source positions with a GPS error of 2m. The estimates are done by the EKF 3 model. Each initial position guess was run 5 times, the average RMS position error for the 5 runs is computed and plotted for each time step. The RMS position error for the excellent, poor, and mediocre initial guess are the red, blue, and green lines respectively.

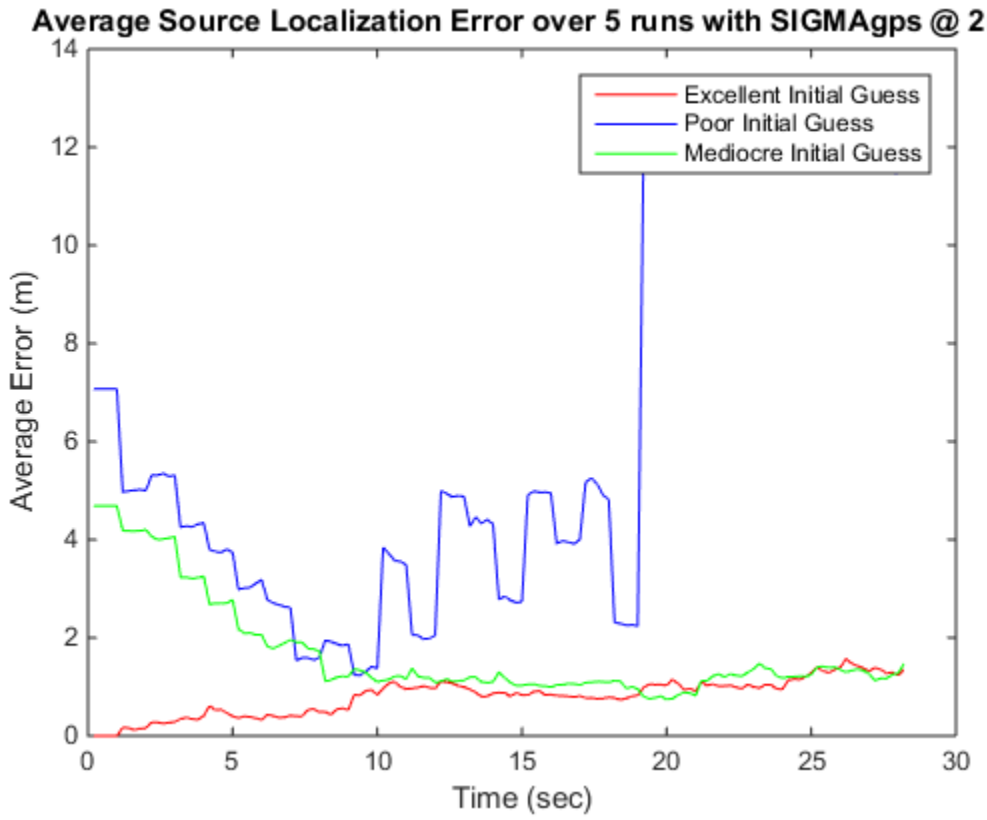

Figure 3.30 Results 3.8 Average Source Localization Error over 5 Runs 2m GPS Error: This figure shows the error in the source localization estimates with varying initial guesses with a GPS error of 2m. The estimates are done by the EKF 3 model. The simulation was run 5 times for each initial position guess. The average error at each time step is computed and plotted. The localization estimate error for the excellent, poor, and mediocre initial guess are the red, blue, and green lines respectively.

Table 3.19 Results 3.8 Table of Tested Values 3m GPS Error

σ_{GPS}	3m
σ_{RadPos}	R:1m B:5m G:3m

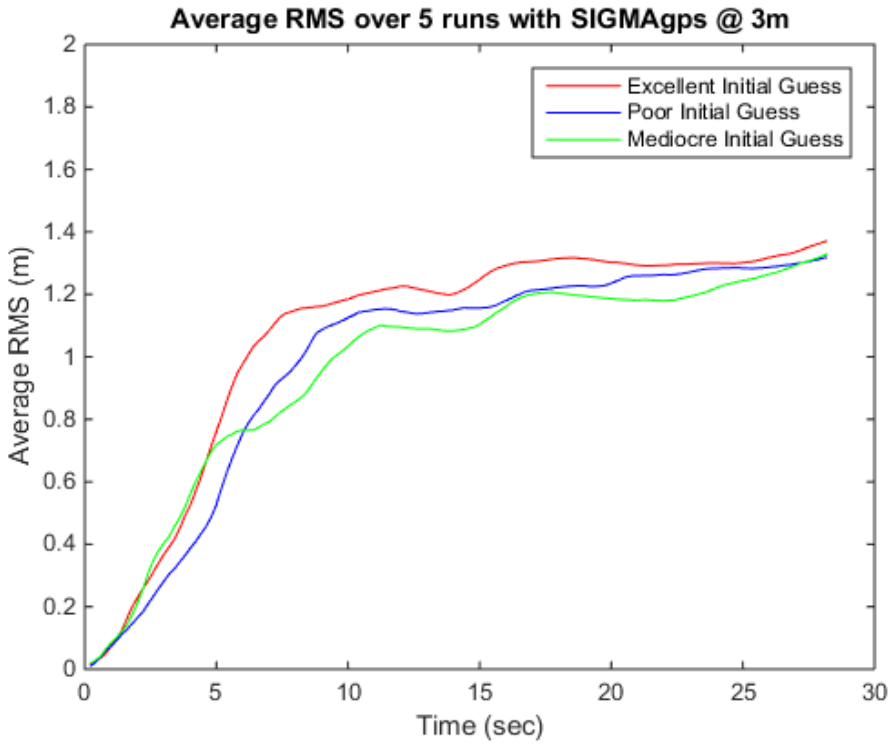

Figure 3.31 Results 3.8 Average RMS over 5 Runs 3m GPS error: This figure displays the RMS position error for three runs with varying initial guesses in source radioactive source positions with a GPS error of 3m. The estimates are done by the EKF 3 model. Each initial position guess was run 5 times, the average RMS position error for the 5 runs is computed and plotted for each time step. The RMS position error for the excellent, poor, and mediocre initial guess are the red, blue, and green lines respectively.

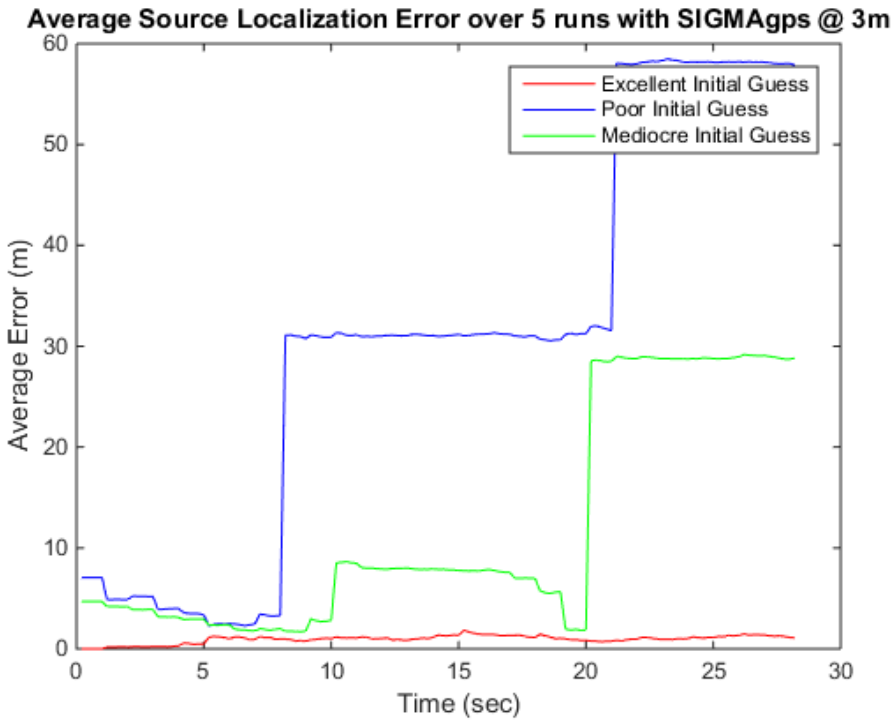

Figure 3.32 Results 3.8 Average Source Localization Error over 5 Runs 3m GPS Error: This figure shows the error in the source localization estimates with varying initial guesses with a GPS error of 3m. The estimates are done by the EKF 3 model. The simulation was run 5 times for each initial position guess. The average error at each time step is computed and plotted. The localization estimate error for the excellent, poor, and mediocre initial guess are the red, blue, and green lines respectively.

3.9 TESTING THE RELATIONSHIP BETWEEN VEHICLE POSITION AND LANDMARK OBSERVATION

In this experiment the relationship between vehicle position and landmark observation is tested. The vehicle flies in two different environments, one with a wall to observe and the other with nothing. Scenarios were run five times each with varying GPS error. Two different scenarios were tested. In the first scenario the vehicle starts at position (5m, 7m, 5m) and flies to waypoint (15m, 7m, 5m) then returns. For the second scenario instead of strafing the vehicle starts at (5m, 2m, 5m) and flies to (5m, 7m, 5m) and then returns back. Instead of side strafing, the vehicle moves forward measuring the same landmarks as it moves. During the flight the vehicle is facing along the y-axis. In one environment, it faces a wall 8m high wall that spans the length of the world. In the other there is no wall for the vehicle to observe. In order to rule out radioactive measurements a source of activity 0 counts per second is placed in the environment and is initially guessed to have zero activity. Table 3.20 and 3.21 shows the default values used for the experiment. Additionally, figures 3.33, 3.34, 3.38, and 3.39 show an example world and actual flight path for both environment types.

Table 3.20 Results 3.9 Measurement Errors

Error Variable for Measurements and R Matrix	Actual Error Value	Value used in R Matrix
σ_{GPS}	varies	varies
$\sigma_{Compass}$	.1rad	.1rad
σ_{IMU}	.1 rad	.1rad
σ_{Gyro}	.1 rad/s	n/a
$\sigma_{Landmarks}$	n/a	1m
σ_{Rad}	n/a	4000 counts/s

Table 3.21 Results 3.9 Estimated Model Errors

Error Variable for Q Matrix	Value Used in Q Matrix
σ_{XYZ}	.02m
σ_{V}	.1m/s
$\sigma_{Heading}$	.1rad
σ_{IMU}	.1rad
$\sigma_{Landmark}$	.5m
σ_{RadPos}	1m
σ_{RadInt}	1 counts/sec

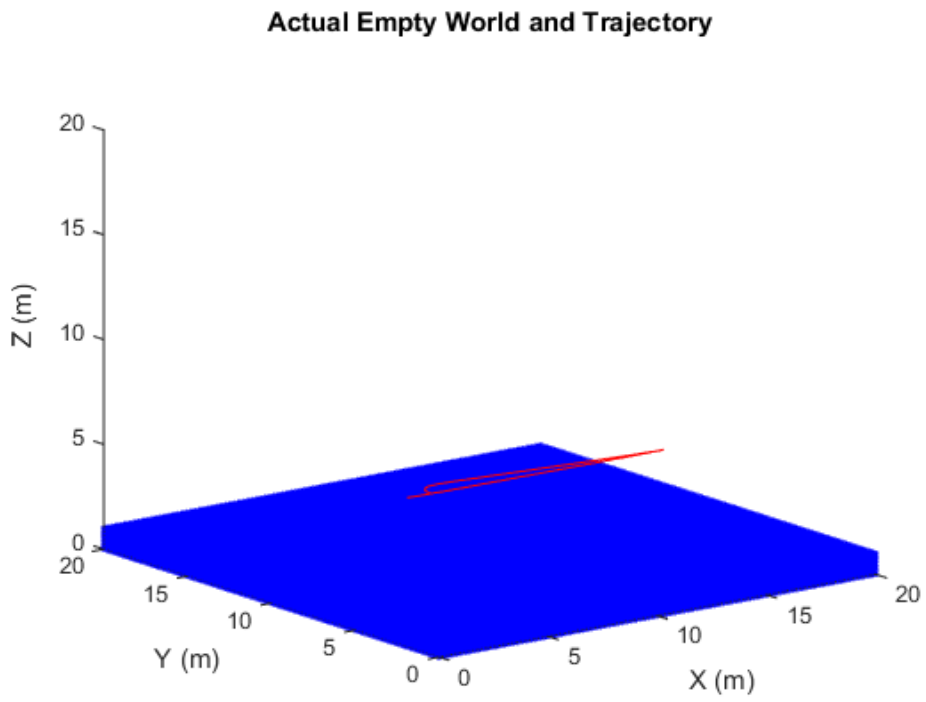

Figure 3.33 Results 3.9 Actual World and Trajectory no Wall Strafe: This figure shows the actual flightpath and world for a strafe test without an observable wall. The simulation is run using the EKF 3 model for estimation. The red line shows the vehicle's flight path starting from the left, strafing right, then coming back. The blue represents the world.

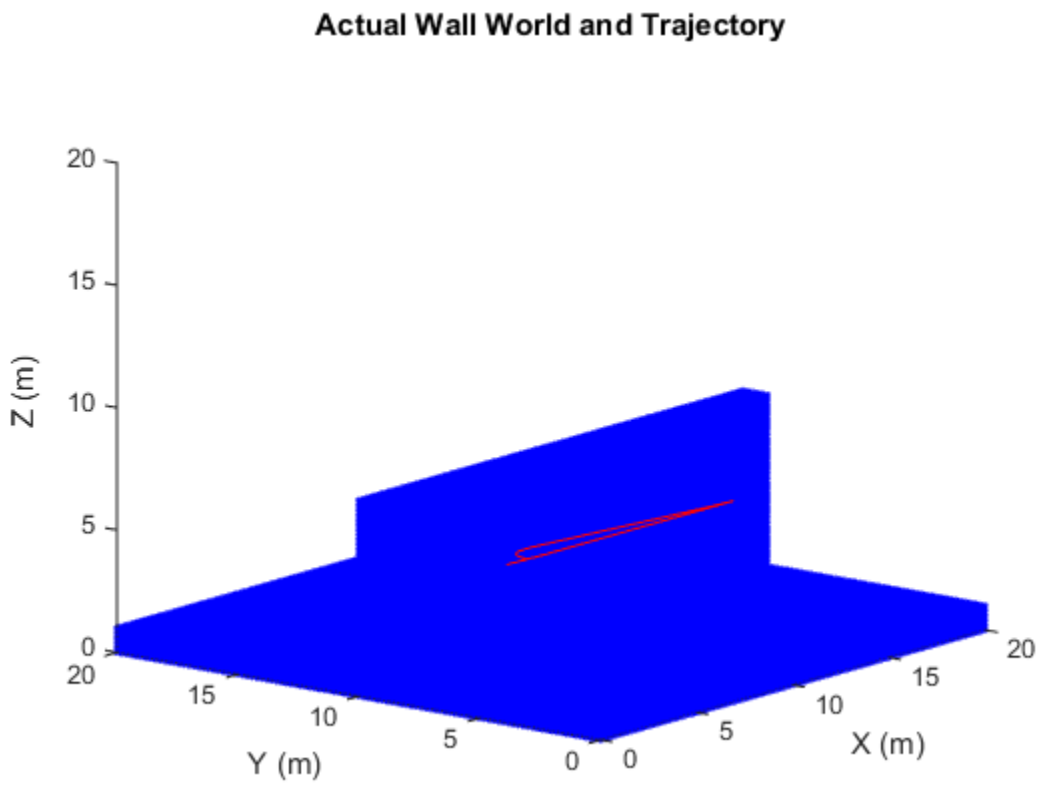

Figure 3.34 Results 3.9 Actual World and Trajectory Wall Strafe: This figure shows the actual flightpath for a strafe test with a wall to detect. The run is done with the EKF 3 model for estimates. The red line shows the vehicle flightpath, the blue represents the world.

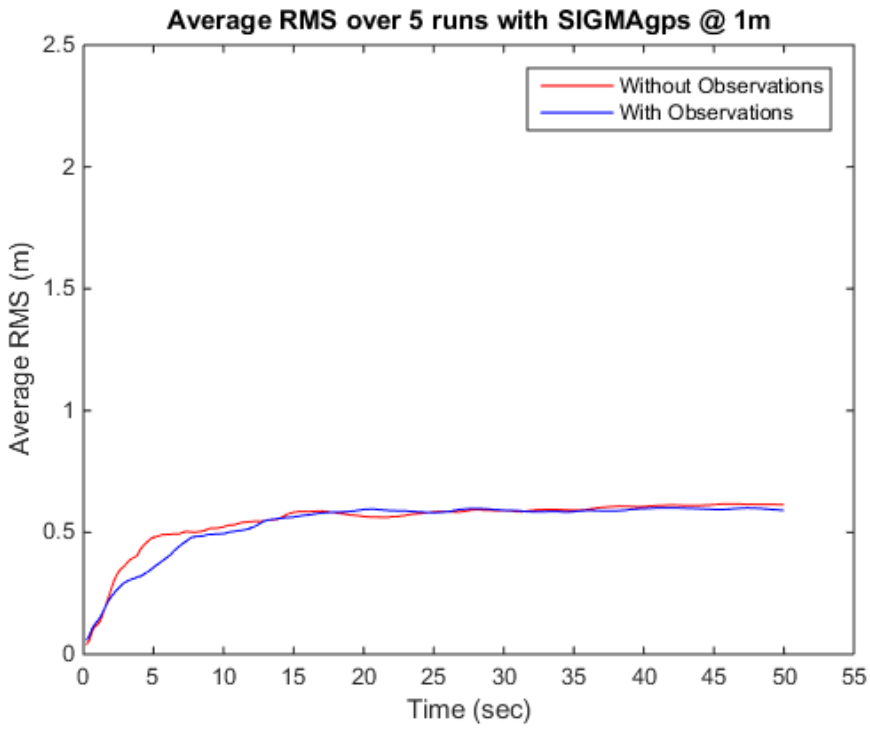

Figure 3.35 Results 3.9 RMS Position Error over 5 runs 1m GPS Error Strafe: This figure shows the RMS position error over 5 strafing runs (lateral motion parallel to the wall) with a 1m GPS error for both scenarios with and without observations. The estimates are done by the EKF 3 model during the strafing motion. The red line shows the RMS position error for the runs without observations. The blue line shows the RMS position error for the runs with observations.

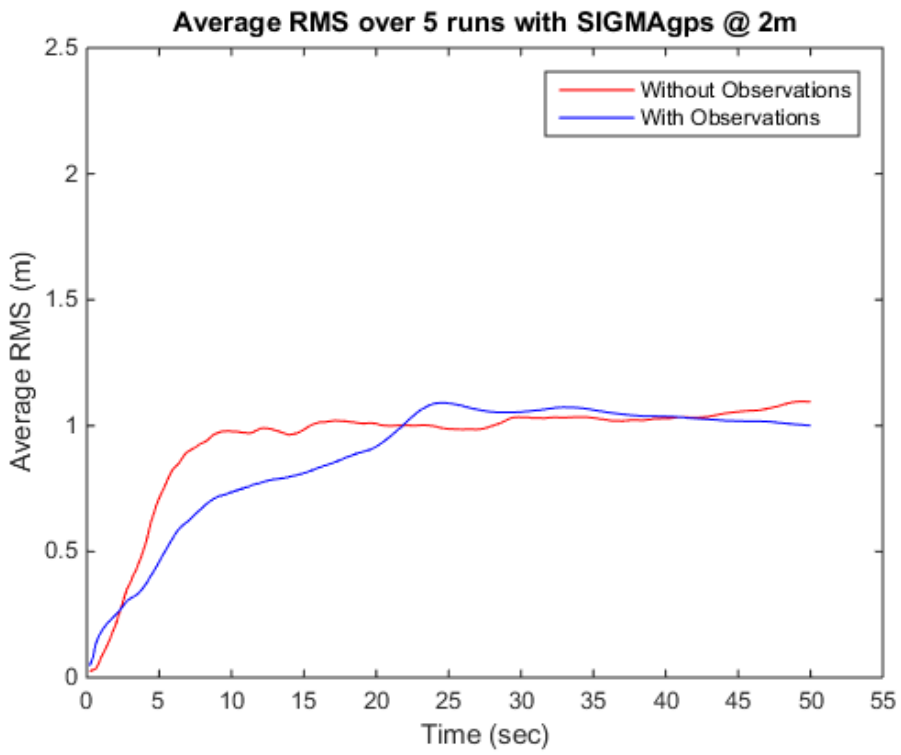

Figure 3.36 Results 3.9 RMS Position Erorr over 5 runs 2m GPS Error Strafe: This figure shows the RMS position error over 5 runs (lateral motion parallel to the wall) for the strafe test at 2m GPS for both scenarios with and without observations. The estimates are done by the EKF 3 model. The red like shows the RMS position error for the esimates without obervations. The blue line shows the RMS position error for the estimates with observations. Here the estimates with observations peak a higher error, however trend to a lower error towards the end.

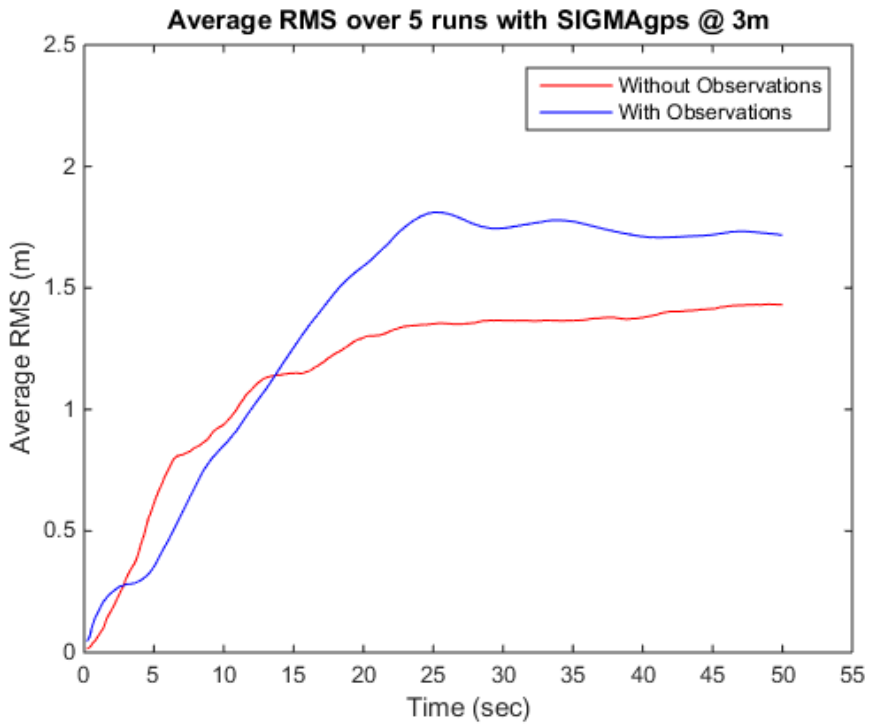

Figure 3.37 Results 3.9 RMS Position Error over 5 runs 3m GPS Error Strafe: This figure shows the RMS position error over 5 runs with a 3m GPS error for both scenarios with and without observations. The estimates are done by the EKF 3 model. The red line shows the RMS position error in the estimates for runs without observations. The blue line shows the RMS position error for the estimates with observations. The runs with observations have significanly higher RMS position error, however as the flight progresses that RMS position error has a downward trend.

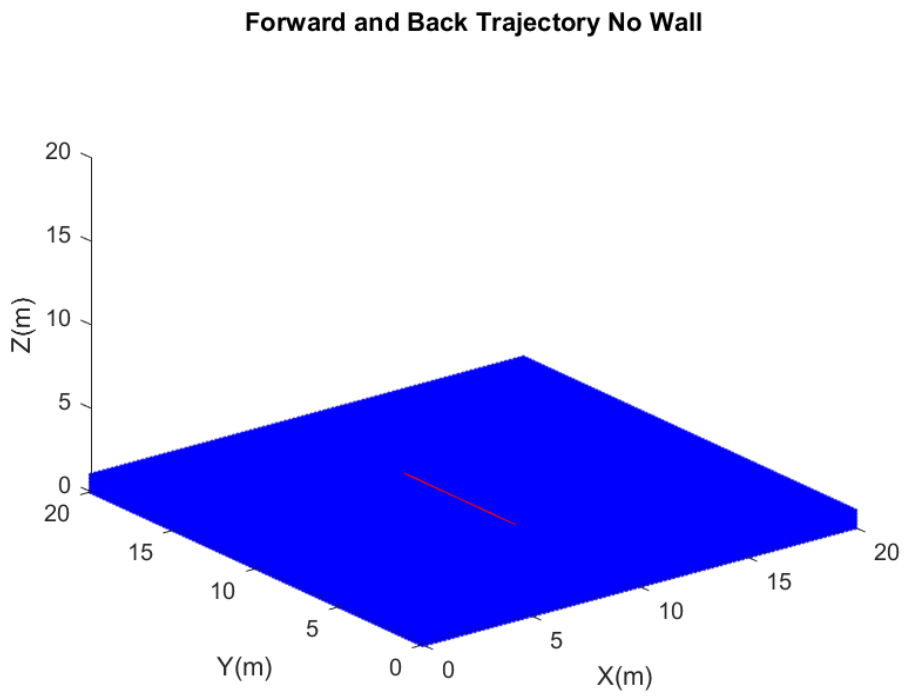

Figure 3.38 Results 3.9 Actual World and Trajectory no Wall Forward/Back: This figure shows the actual flightpath and world for a vehicle flight in a forward/backward motion, with no observable wall, high enough to not detect the floor. This run is tested on the EKF 3 model and at varying GPS errors. The red line shows the vehicle flightpath and the blue shows the actual world.

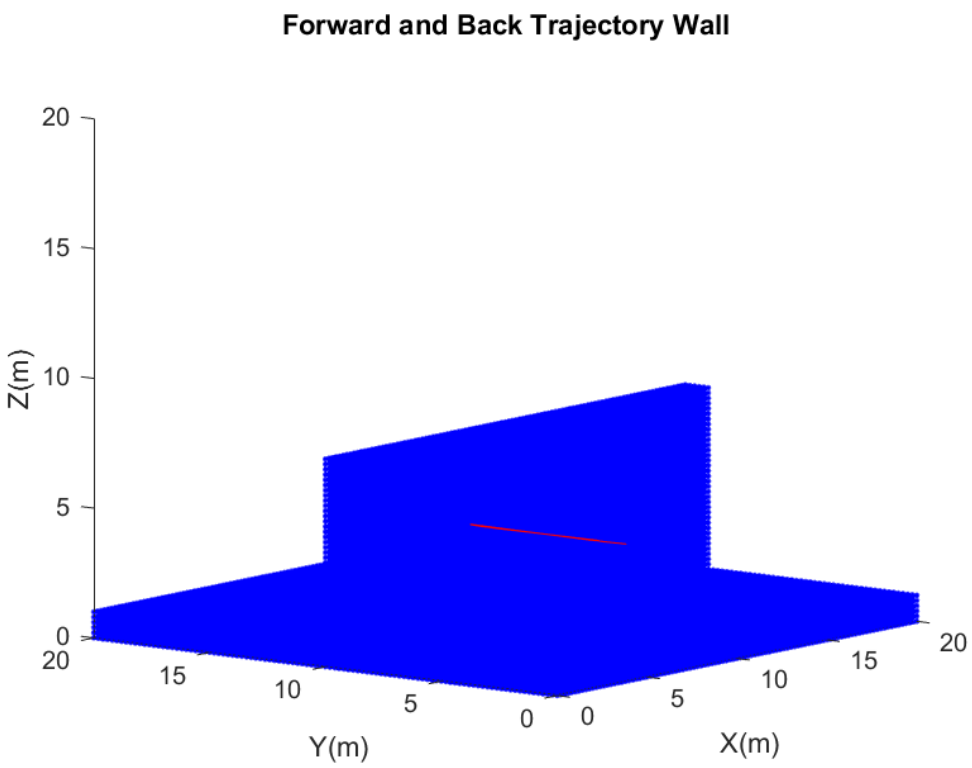

Figure 3.39 Results 3.9 Actual World and Trajectory Wall Forward/Back: This figure shows the actual flightpath and world for a vehicle flight in a forward/backward motion towards an observable wall. This run is tested on the EKF 3 model and at varying GPS errors. The red line shows the vehicle flightpath and the blue shows the actual world.

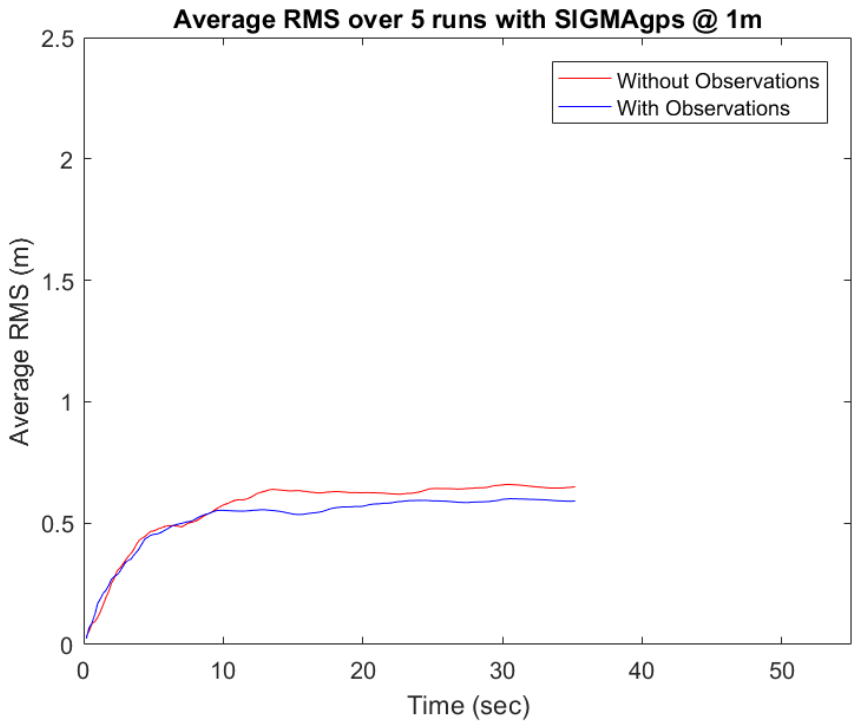

Figure 3.40 Results 3.9 Average Position Error RMS over 5 runs 1m GPS Error Forward/Back: This figure shows the RMS position error over 5 forward and back motion runs with a 1m GPS error for both scenarios with and without observations. The estimates are done by the EKF 3 model during the motion. The red line shows the RMS position error for the runs without observations. The blue line shows the RMS position error for the runs with observations.

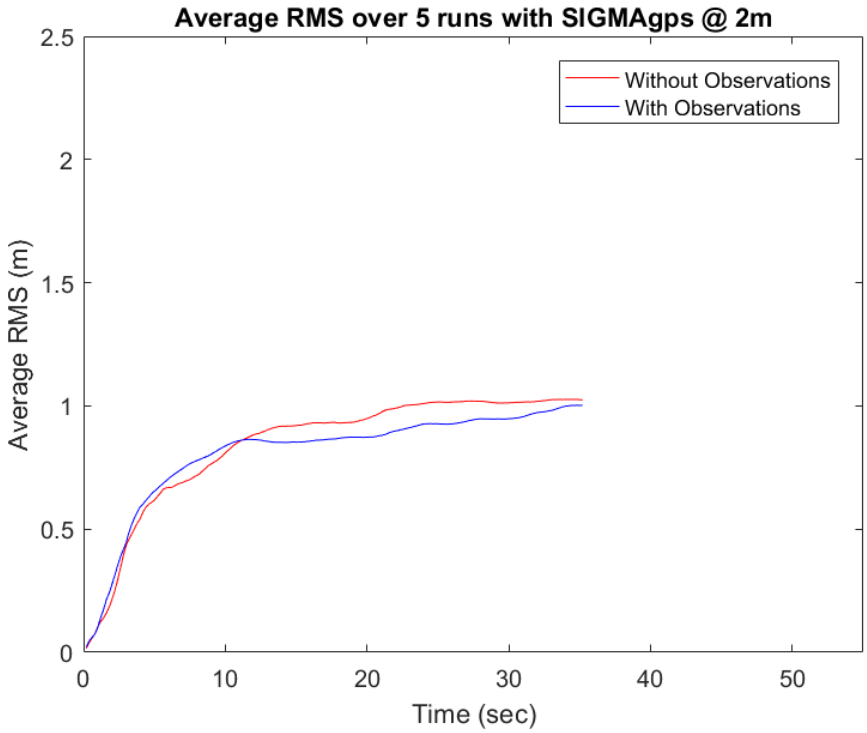

Figure 3.41 Results 3.9 Average Position Error RMS over 5 runs 2m GPS Error Forward/Back: This figure shows the RMS position error over 5 forward and back motion runs with a 2m GPS error for both scenarios with and without observations. The estimates are done by the EKF 3 model during the motion. The red line shows the RMS position error for the runs without observations. The blue line shows the RMS position error for the runs with observations.

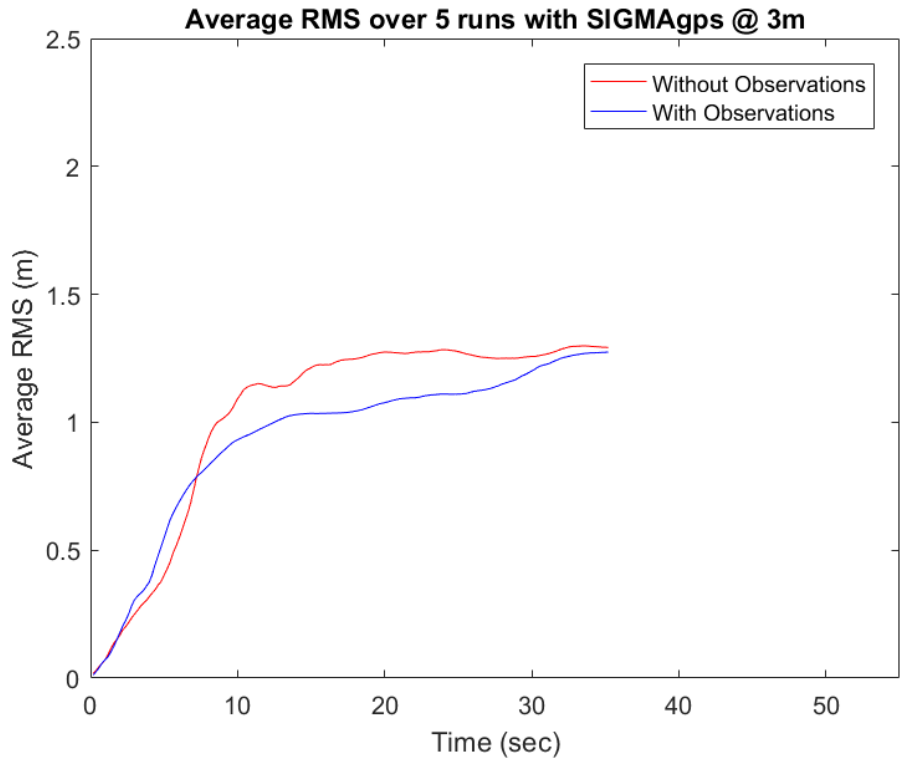

Figure 3.42 Results 3.9 Average Position Error RMS over 5 runs 3m GPS Error Forward/Back: This figure shows the RMS position error over 5 forward and back motion runs with a 3m GPS error for both scenarios with and without observations. The estimates are done by the EKF 3 model during the motion. The red line shows the RMS position error for the runs without observations. The blue line shows the RMS position error for the runs with observations.

3.10 VEHICLE LOITER AND SOURCE LOCALIZATION UPTDATE TIME LAG TEST

This experiment was done after discussions with lag times on the vehicles ability to update the source position after moving into a location with good data. Each graph represents five runs where the vehicle loiters at a different location away from the source. The four locations in order of presentation are:

Location one – (10m, 10m, 4m) Total Distance: 2m

Location two – (10m, 10m, 5m) Total Distance: 3m

Location three - (7.5m, 7.5m, 5m) Total Distance: 4.64m

Location four – (5m, 5m, 5m) Total Distance: 7.68m

The source itself has an activity of 4000 counts per second and is location at (10m, 10m, 2m). The vehicle loiters at one of the above position for 40 seconds of time, taking in counts every second and attempting to localize the source.

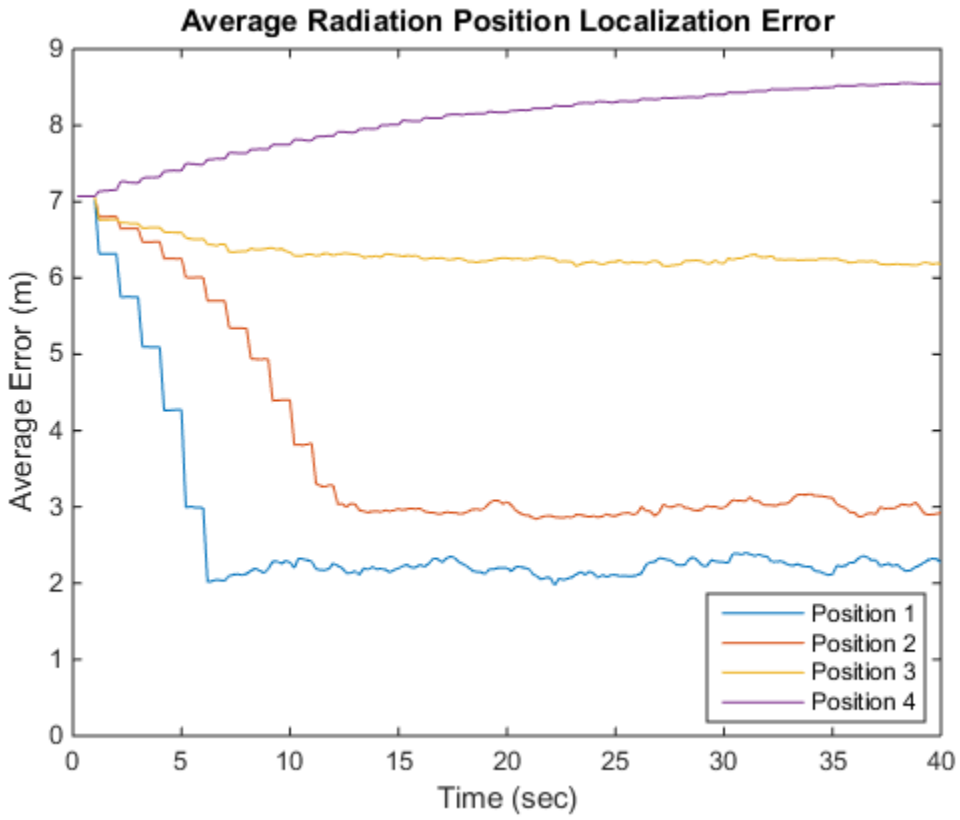

Figure 3.43 Results 3.10 Average Radiation Position Error: This figure shows the estimated radiation localization error averaged over 5 run for differing loiter positions. The estimates were done by the EKF 3 model. Position 1 (blue) loiters at (10m, 10m, 4m). Position 2 (red) loiters at (10m, 10m, 5m). Position 3 (yellow) loiters at (7.5m, 7.5m, 5m). Position 4 (purple) loiters at (5m, 5m, 5m).

CHAPTER 4: DISCUSSION

4.1 OPENINING DISCUSSION

Drones are becoming an important tool today for multiple tasks. We want to pursue drone applications in the context of hazardous environments, particularly a radioactive environment. The general idea for now is that an additional on-board processing unit, either micro controller or more sophisticated computing unit such as a raspberry-pi would be carried as a payload. Algorithms were developed in order to process incoming obstacle and radiation data into meaningful and intuitive forms for map building and source localization purposes (and can be generalized to localization of any object with properties that can be detected by a sensor or combination of sensors). The results of the algorithm should be in such a form that they allow for the pilot or autopilot to make navigation decisions. The algorithms developed could be implemented in the on-board processing unit, which communicates to the vehicle flight controller. In this way, the vehicle flight controller does not require any software modifications, allowing for the studied work to be implemented on different UAV platforms.

This work assumes no a priori vehicle position knowledge, seeks to limit the use of 'accurate measurements' and attempts to solve a problem consisting of vehicle localization, map building, and radioactive source localization. Published work provides satisfactory results for each criterion individually, but falls short of combining all three into one system. Looking at "Synchronous Radiation Sensing and 3D Urban Mapping for Improved Source Identification" (Christie, Stiltner et al. 2014), this group of researchers make no use of a Kalman Filter at all, taking raw measurement data, and combining radiation measurements with a GPS measurement of the location the measurement was taken. This relates very similarly to the way the first two algorithms work, where the Kalman Filters simply process the radiation measurement data where it was collected, rather than fill in the gaps where radiation measurements were not taken. Additionally, referring to their figure 3, which displays planned

and actual flightpaths, it is uncertain whether this actual flightpath was how it appears in figure 3(b) or simply reflects the GPS measurements made during vehicle flight. If the flight path measurements are solely taken from GPS measurements, then simply supplying the raw data from measurements could be quite inaccurate. Also this proposed work is for urban mapping; however, their vehicle flies at a relatively high altitude (20m – 25m) where obstacles are a minimal issue, but also where sensor data and even visual access to all objects of interest could be limited by distance and visual complexity.

Other research shows payload construction for radiation sensors for UAV applications, but processing of data and results seem to be somewhat of an afterthought. In "Designing a Radiation Sensing UAV System" (Cai, Carter et al. 2016) a prototype Radiation Sensing UAV system was developed. Their general payload design and functionality is very similar to the sensor modeled in the simulation. Their chosen detector Teviso RD3024 measures beta and gamma radiation, along with X-rays. Beyond collecting the radiation data, this group attempts to use the data to determine safe walking paths and influence vehicle motion. Algorithms described do not appear include the use of any estimators or a Kalman Filter. The work ties again very closely to the research presented here; in contrast to (Cai, Carter et al. 2016), we use the (Extended) Kalman Filter heavily for algorithms developed, specifically for integrating raw data from multiple sensor types. Additionally, the UAV applications appear to be for a high altitude flight, where obstacles are again at a minimum but also limiting radioactive readings from low activity sources.

Moving away from radiation-focused goals, other relevant work in spatial representation and obstacle avoidance includes "Obstacle Avoidance and Active Disturbance Rejection Control for a Quadrotor" (Chang, Xia et al. 2016) where the research is focused on the controls of the vehicle in terms of obstacle avoidance. The authors seem to minimally address some relevant issues, which are vehicle localization and obstacle detection, assuming that the vehicle position is known and assuming that the object can be measured by the vehicle sensors. Beyond stating this, there is not much else other than

successful obstacle avoidance. The research presented in this thesis attempts to address a more realistic scenario where the vehicle position and obstacles are not accurately known.

Both (E)KFs 1 and 2 represent the environment via cubes, and estimate the occupancy of that space. "Real-time Autonomous UAV Formation Flight with Collision and Obstacle Avoidance in Unknown Environment" (Cetin and Yilmaz 2016) which has an emphasis on formation flying also hold relevance from both obstacle avoidance and environment representation. Again the researchers here touch lightly on obstacle sensing and vehicle localization. However, the similarity of world representation as compared to the representation in the first two algorithms presented in this thesis are apparent. Similar to a cube method, the researchers instead have a 2-D grid, the values in the grid instead of occupancy represent the highest point in that specific space. This however runs into similar issues as the two algorithms developed here run into.

Overall there is a great deal of work done along the lines of the research presented in this thesis, however the research present contains its own genuine touch in the form of solving a larger problem with respect to issues faced when piloting a vehicle in a unknown environment. Those issues consist of vehicle localization, world mapping/estimation, and the addition of radioactive detection and source localization. It attempts to provide a solution to these problems by implementing a series of routines and using a well-respected estimation process known as the Kalman Filter.

4.2 KF 1 SIMPLE WALL AVOIDANCE DISCUSSION

These results show a very simple scenario for the first Kalman Filter, specifically looking at the proximity sensing and the ability of the vehicle to avoid obstacles. Looking at figures 3.1 and 3.2 which display the vehicle trajectories and both the actual and estimated world, we can see a successful avoidance procedure. As the vehicle moves from its initial position (2m, 5m, 3m) and flies toward (9m, 5m, 5m) the vehicle almost immediately increases its altitude. It gets close to the wall before making a quick backwards maneuver to avoid the top edge of the wall. The initial motion of the vehicle seems to

point towards the fact that initially that portion of the wall could not be measured by the sensors as the vehicle was too close and too low for a portion of the beams to measure that point. Once the vehicle gets high enough to begin measuring the top portion of the wall, the obstacle avoidance routine will redirect the vehicle to avoid that segment. After the maneuver is made, the vehicle smoothly reaches the final waypoint destination.

In figure 3.2 there are a few things to point out. The first relates to the world estimate, where there are points to the right side of the wall that are considered occupied. What happens sometimes is that there can be false positives for occupancy because when the measurements are made – they are made relative to the vehicle. However, the vehicle's position is only known through its position estimates, meaning that poor position estimates when measurements are made can cause poor world estimation. When looking at figure 3.2 it becomes clear that the position estimates at some times are quite poor, particularly in situations when the vehicle makes a sudden maneuver. These poor estimates are likely causing measurements in cubes where the wall does not exist.

Again looking at figure 3.2 a trend in the filter can be seen where it does well for longer motion trends of the vehicle. When the vehicle moves in a different direction, the KF tends to estimate the vehicle position poorly for some time after the new maneuver. In general, it appears the KF does much better with a more constant vehicle velocity. This is something that can be seen often.

Overall, the results are quite good for this scenario, where the occupancy cubes estimated by the KF allow the vehicle to determine occupancy in the world. The algorithm identifies volumes of space that can be deemed dangerous, allowing for simple obstacle avoidance and safe trajectory changes.

4.3 KF 1 GRID PATTERN FOR RADIOACTIVE MEASUREMENTS DISCUSSION

The purpose of this experiment was to visually quantify the output of radiation estimation from the KF. With one of the goals of localizing a radioactive source, a top down view was envisioned in order to get a sense for where a larger number of counts were being acquired. The vehicle flies in a grid pattern as seen in figure 3.4. The actual world is displayed as well; in this case, to isolate the radiation component of the filter there are no obstacles and the vehicle flies high enough to not take ground measurements. As the vehicle is flying in the grid pattern, it produces results as shown in figures 3.7 and 3.8, where Fig. 3.7 is a cleaner display of only the radiation count estimates. Figure 3.8 is the same display with estimated trajectory overlay. This is useful in understanding why the estimates are where they are. As explained in section 2.5.1 the KF only estimates in cubes where measurements were made. In reality, due to errors in the estimated position of the vehicle, it is more correct to say that cubes are only estimated in regions that the algorithm identifies as having been measured. This is a distinct difference (in comparison to some previously published work) as it is subject once again to the ability of the KF to estimate the vehicle position.

Overall the top down view of the data in figures 3.7 and 3.8 show agreement between count rate and estimated source position as expected. It can be seen that the circles closer to the source (as marked by the red X) are larger than the circles on the far side of the space. One disadvantage of this KF design regarding radiation measurement estimates is that a large distance must be traversed in order to get multiple measurements and estimates to make a decision as to where the source may be. If this information is to be used in planning the vehicle flight path, the information would have to be processed slightly to determine a potential location of the source. For example, a pilot or operator could potentially look at the top down and estimate the source position, but a computer would need to perform further data processing to quantify that position.

4.4 KF 1 ADVANCED WORLD WITH OBSTACLE AVOIDANCE LOW ERROR DISCUSSION

For these results, the vehicle was placed in one of the most complex worlds tested. When the environment contains multiple objects, the world estimate and obstacle avoidance results can vary. We wanted to test and look at how well the KF, and the obstacle avoidance would work in a harder environment. While conceptually the experiment does not appear to be more complex, with more than just a wall and floor being sensed the, gradient (of the effective obstacle occupancy function) may fail to properly identify safe areas to navigate through. When the gradient for course correction is computed, it uses all the environmental estimate information. For example, from the start, the vehicle tends to head away from the tree's base. This is because since the vehicle detects it, that occupied space contributes to a "force" that will tend to push the vehicle away from it. As it approaches the dip in the ceiling, the gradient that is computed forces the vehicle down and slightly in the negative x-direction from the potential collision site. Looking at figure 3.10 and specifically, the world occupancy estimates there are several locations that the KF estimates to be occupied that which when looking at figure 3.9 or 3.12 are clearly not occupied. This is again due to proximity measurements being made using the estimated vehicle position. This marks occupancy of areas that are not actually occupied. This in some way helps the vehicle's obstacle avoidance slightly: areas explored when vehicle position estimates are very uncertain could potentially be a risk to the vehicle, but the effect of the "false positives' in obstacle occupancy can provide in some cases a redundancy, or "padding effect" in identification of obstacles.

Elaborating on this padding effect, figure 3.10 shows the vehicle goes down far lower than was truly required. This is likely due to false positives estimated in the occupancy of cubes. Where the ceiling dips down, smaller circles can be seen at the edge of the dip. As the vehicle makes its initial descent it begins to detect the edge of the opening. The edge is initially lower than it is in reality and causes the vehicle to descend further before it gets closer. As it gets closer, it begins to measure the area with false occupancy and notices it is clear. Once the KF world estimates go to zero or below zero,

the space is deemed safe for travel, and the vehicle travels to the final point without safely. Nonetheless, in figure 3.10, remnants of the incorrectly occupied space still exist to the left and right of the vehicle flightpath.

The radiation data is shown in figure 3.12 where the source was located close to the starting location of the vehicle. The radiation count rates there are much larger than on the other side towards the final waypoint.

Overall the KF and obstacle avoidance performed well in this environment. Poor position estimates lead to a padding effect, which can be both good and bad depending on the circumstances. Radiation measurement estimates can lead a pilot or operator toward location of the source(s).

4.5 KF 1 GENERAL DISCUSSION

The first Kalman filter worked well, especially for vehicle position estimation. It will be nearly impossible to get the exact location of the vehicle, and as such the estimates provided by the KF are good enough to use for proximity and radiation sensing. Recall the general design of the first KF. It estimates only 1000 cubes that form a total volume of 1000 meters cubed. The vehicle is limited to fly within this space. It is possible to increase this space to be 20m in length width and height, however at the current cube size that makes 8000 cubes. At 1000 cubes the KF state vector length is 2009. This makes for massive computations. This destroys the feasibility of using this filter for on-board real-time obstacle avoidance, except if the environment representation within the KF is kept small. Using Matlab's Tic and Toc functions on a desktop processor rated at around 96 Gflops, a computation time of over 1 second for the KF process was determined. An on board processor, rated much lower, with around 24 Gflops is likely to take four times as long. As a result, the computational limitations will reduce the usefulness of the algorithm.

Since the filter design estimates cube occupancy, it allows the filter to identify locations unsafe for flight. With that information it is then easy to determine a potential collision and navigate around

the occupied space. Despite being good for navigation, this same design also poses as one of the KF's largest flaws, which is limiting the space the vehicle can fly into a 10 meter by 10 meter by 10 meter space. Even with a size of 1000 meters cubed the computation time is typically over 1 second on a fast desktop computer. The computation time could more than double on a slower processor. The space and time issues really needed to be addressed to make it a more feasible option.

Another major downfall of this algorithm was the lack of connection within the KF between the different components of the filter design, i.e. some connection between vehicle position and obstacle and radiation estimation. For example, the filter clearly sees a relationship between the vehicle velocity in the x-direction and the vehicle position in the x-direction. This is because in the design of the filter, it clearly uses the velocities to increment the vehicle position in the transition model. Nowhere in the design is there a dependency on the vehicle states and the other state variables related to occupancy and radiation. This is something that would be desirable, since in principle, reliable measurements or estimates in one variable would ideally improve estimates in other variable(s).

4.6 EKF 2 LOITER RESULTS DISCUSSION

In these tests we are looking at the EKF 2's ability to measure and estimate the surrounding environment in a loiter situation. The vehicle simply holds its position as best as the autopilot allows around the position (10m, 10m, 2m) and faces 45 degrees above the x-axis. This means that the vehicle sensors should easily pick up on the base of the tree as seen in the real world figure 3.12. The EKF 2's performance can be seen in figures 3.13 and 3.14 which show the estimated world and the position error RMS respectively. Since the radioactive data is only useful in contrast to other points, radiation estimates are not displayed.

Here the EKF 2 does not do too poorly in a loiter scenario, we see occupancy in the world estimates where the wall and tree base are, however the gap between the wall and the tree is not picked up on. The area would not be an ideal flight location, but would still be useful to know that the

space is in fact clear. The error RMS is quite good here, but it is important to note that the EKF estimates do better in a uniform motion scenario, and with the vehicle not moving it is expected to have relatively low error in the EKF estimates.

4.7 EKF 2 GENERAL DISCUSSION

The second design implemented in an Extended Kalman Filter form came about due to some of the weaknesses present in the first design. Those weaknesses are again spatial limitation, computation time, and lack of state variable relationships. An attempt to address each weakness was made in designing the second EKF.

One of the most radical changes made is to the transition model which can be reviewed in section 2.5.2. Now, the world that the EKF attempts to estimate follows around the vehicle as it flies. The idea is now that the transition model depends on vehicle states such as velocities, the vehicle velocities cause a shift in the information. This design simultaneously addresses two of the weaknesses. It firstly addresses the relationship concern. Looking at the Kalman Gain matrix and the Covariance Matrix – there are clear relationships between the world states and the vehicle velocity states. The second weakness is that the limit of the vehicle's flight has been lifted. Since the EKF world moves, the world data is copied and pasted into another larger and more permanent world which can be referenced as the vehicle passes back into familiar space. The EKF only estimates the world within about 10 meters of the vehicle. It is not heard of to have quadcopters reach speeds of over 50 mph, however in a hazardous environment where mapping and source identification is a goal, it is unlikely to travel with speeds greater than 10 mph. The potential limitation of the vehicle's ability to "react" quick enough would depend on a few factors, computational speed and vehicle speed. With vehicle speed being relatively minimized by obstacles and localization goals the 10m estimation boundary is likely "good enough" for obstacle avoidance purposes.

The third weakness which was the computation time was addressed by changing the cube sizes of the EKF world. Instead of cubes with dimensions of 1m, they were changes to 2 meters. This means that it takes 125 cubes to represent occupancy of the 1000-meter cubed volume around the vehicle. An additional 125 cubes are used for radiation measurement estimates. This makes a state vector with total length 259. This is almost a tenth of the original size - drastically improving the computation time down below .2 seconds as measured on the same processor as the above KF time in section 4.4.

There are some major downfalls to this EKF. The first is that the overall results of the EKF are very poor. The output itself vaguely resembles the environment and at times the errors in the estimate can blow up. Beyond loiter results, any motion causes massive issues with the results. The overall failure of the filter likely comes from some of its design components. The 8 meters cubed cubes (2x2x2) encompass a lot of space. This means that the resolution of the world has been greatly reduced. Previously in the KF version 1 only 1-meter cubed space was blocked for safe flight, now a much larger space is estimated meaning even if an object of half a meter cubed is in that space, the entire space will be measured/estimated occupied. This greatly reduced the usability of the EKF. This increased volume also exacerbates a bad measurement. Since measurements are made relative to the vehicle and the vehicle position estimates are not 100% accurate, these measurements can be in the wrong cube. This is much like the first KF where visually there is false occupancies occurring. In the case of this EKF, those false occupancies have a greater impact. When attempting obstacle avoidance at times the vehicle can come to a halt. This is a major flaw to the EKF, where the occupancy estimates are really not usable.

Another factor contributing to the poor performance could also lie in the transition model. With the information essentially being averaged as it shifts there is a lot of room for error. The direction of shifting depends on the estimated vehicle motion, so any time the vehicle is estimated to shift in the wrong direction the shift will also occur in the wrong direction. Additionally, the estimate may shift too

much or too little far too often causing basically a blurring of the information. This effect is shadowed by the size of the cubes – however remains present.

With difficulties getting solid world results from this filter, it wasn't tested very much. Its original design on paper seemed quite solid, but in the end did not work well. There are some improvements that could be made to this EKF, for example changing the cube dimensions back to 1m. This would significantly help with bringing back a better world resolution. Perhaps to address the computation time issue; the total volume estimated could be reduced down to at most 5 meters. If it was changed to 7m length, width and height volume the total number of cubes would be reduced to 343. This would be a reasonable compromise as one of the strengths of the first KF was the resolution of the estimated world and how well it worked for obstacle avoidance.

4.8 EKF 3 SIMPLE WORLD OUTDOOR/URBAN RESULTS DISCUSSION

The purpose of both results sets for the outdoor and urban world is to introduce the general results of the filter. We are asking the question, will the EKF perform in both an outdoor and urban environment. When viewing the overall estimated world and trajectories shown in figures 3.16 and 3.20 the overall EKF performance on vehicle and world estimate are quite good. A major part of the map building component is the vehicle estimation. Without some form of accurate position estimates the map can quickly go from decent to poor. Once again the error in the estimates for both cases are significantly lower than that of the raw GPS measurements. This is significant for producing the map. All measurements are again relative to the vehicle, and as such even an accurate measurement is subject to error in vehicle estimates. Over this EKF does a good job at estimating the most essential part – which is the vehicle estimates.

Moving on to the outdoor world estimates shown in figures 3.16, the outdoor environment does not look too bad. While visually it is tough to see the estimate world, it can be known that portions of the wall were successfully detected as the vehicle climbed over the wall to avoid it. As the vehicle

continues to fly around the estimated world continues to be generate but is limited to observations done by the vehicle. It can be seen that there is some object beginning to be formed that resembles the tree placed near the wall, however the vehicle flies below the tree limiting its observations to the base and under part of the tree. In this example the vehicle is close enough to the ground as to start taking ground measurements. The world generated is sufficient for implementing a navigation and obstacle avoidance system.

The urban environment is another example of a successful EKF world estimation. In this case the vehicle is too high to detect the ground. In order to better resolve the building locations visually, figure 3.19 z-axis is scaled to exclude the ground. Figure 3.20 is the results of the estimated world. It performs quite well – in fact there is even an alley between the first two buildings on the left that appears to have been seen by the EKF world. These results however are not without some limitations. Looking closely at figure 3.20 it can be seen that some landmarks exist in areas where they should not. This can drastically affect the ability of the vehicle to navigate, especially with avoiding an obstacle not truly there. There will be more discussion on this later.

The radiation results shown in figures 3.18 and 3.23 are also quite promising. With the new model explained in section 2.5.3 for the radiation, the EKF is now attempting to estimate the source's activity and location. Overall this a total of four variables, x, y, z, and activity which limits the estimate accuracy. In the figures the actual distance between the vehicle and the source is also plotted. Since the counts at a distance are dictated by a one over r-squared rule, this means that the closer the vehicle is to the source, the lower the uncertainty in the data is – allowing for increased accuracy in estimation updates. Where measurements vary little during motion, the estimation process will prove difficult. The source size is arbitrary, however as the derivative w.r.t. position of the count rate tends to zero, the filter's ability to estimate the position and activity will be drastically reduced.

A few other figures are presented to look at the general performance of the vehicle estimation. Figures 3.16 and 3.20 show the RMS error in vehicle position estimates along with GPS measurement error. It can be seen in both cases that the estimates are far improved from the measurements. Additionally, figure 3.22 shows the heading estimates for the vehicle during the urban flight. This figure serves two purposes, first to show the results of the heading estimates and the second is to show that the vehicle faces the direction it is flying in as it navigates to different waypoints.

4.9 EKF 3 WALL AVOIDANCE VERSUS GPS MEASUREMENTS DISCUSSION

The purpose of this investigation is to look at the obstacle avoidance as the position measurements taken by the EKF diminish in quality (increase in uncertainty). Figures 3.24 and 3.25 are examples of the environment and trajectories. The first figure represents a successful obstacle avoidance run, while the second shows what a failed run may look like. In the case of a failed run the trajectory is checked to have passed through occupied space. At each GPS error setting the simulation is run 20 times. Figure 3.26 shows the final results of these runs. The avoidance routines work well up to an error set to 3m. The error is slightly misleading, as the measurements taken as GPS are not truly longitude, latitude and altitude – instead they are assumed to have already been transferred into the inertial x, y, z coordinate system. The error is set to a single dimension of measurement so in reality, for a 3m error in each coordinate, the total magnitude of error is around 5.5 meters. For comparison, the 2-meter error is around 3.4 meters. Overall the obstacle avoidance is done by checking a line from the estimated vehicle position to the desired waypoint for any landmark. If there is a landmark in between then the vehicle will redirect. With poor estimates it is likely to not see landmarks in between the vehicle and the desired waypoint. The EKF depends heavily on GPS measurements and it is quite unlikely the vehicle will be flown in an environment with such high GPS error.

4.10 RELATIONSHIP BETWEEN VEHICLE POSITION AND RADIATION DISCUSSION

These tests look at how vehicle position and radiation affect each other within the EKF. The test was run by setting the GPS error to a fixed amount, and adjusting the accuracy of the initial EKF guess of the radioactive source. Each guess scenario is run 5 times in order to get a trend of the RMS position error RMS and the radiation error. The results shown are the average value at each time set for the values. By adjusting the initial error, the EKF process will modulate the residual (the difference between the estimated and actual measurements) causing different amounts of correction to the estimates. In order to look at the position effect on the radiation this test is repeated a total of three times with varying GPS error. This again increases the residual for the position estimates – thus changing estimates.

Looking at the first set of results for the 1m of GPS error, all three position estimates perform similarly regardless of initial guess on the source location. If the effect of the radiation measurements is significant then there are two reasonable outcomes. The first is that if the effect is positive, that is to say it improves estimate quality, then the runs with an excellent initial guess should outperform the other runs in position estimates. To further elaborate, the mediocre guess runs should be behind the excellent, but ahead of the poor guess runs. Figure 3.27 shows that by the end of the run the excellent guess performs the better than the poor guess, however the mediocre guess by the end performs the best. In general, each set for the different guesses perform relatively similar, such that there is probably no significant difference in performance. As the GPS error is increased in figure 3.29 the difference is much more significant. The runs with the excellent initial guess perform the best and the mediocre guess performs worse by the end of the simulation. Again, these values are all relatively close – in fact each line falls within the mean absolute deviation of each other (MAD). The MAD lines are not displayed in order to clean up the graphs. The results for the final set of runs where the GPS error is at 3m changes the order of performance yet again. At the end of the simulation runs the excellent guess

performed the worst and the poor and mediocre guess ended at about the same level of performance, however for a significant portion the mediocre guess performed the best. It should be pointed out that they are all still performing relatively close to each other.

The radiation estimates are significantly different. The first set where the GPS error is at 1 meter gives expected results where by the end of the runs they stabilize with about the same amount of error (FIGURE 3.28). As the GPS error increases, the results begin to vary. Figure 3.30 shows that a poor initial guess drastically reduces the accuracy radiation source position. Again in figure 3.32 where the GPS error is at 3m the estimates for both the poor and mediocre guess perform quite poorly. Interestingly in each case the estimates begin to improve, then suddenly seem to degrade. They then tend to remain constant for some time before changing again if they do. This is a significant difference as the GPS error increases, and in fact could be quite problematic for the overall confidence in the EKF's ability to estimate the radioactive sources. The overall results showing the difference in vehicle position estimates are nearly insignificant. They are so similar and there seems to be no real order of performance as the GPS error increase. If there is an effect it is unlikely to be measurable. The radiation results and how they are affected by the GPS measurements are however quite shocking. The higher GPS error nearly tears the viability of the current radiation model to shreds. It is however quite unlikely for the drone to be flown in an environment with such high GPS error.

4.11 RELATIONSHIP BETWEEN VEHICLE POSITION AND LANDMARK OBSERVATION DISCUSSION

For these results we are looking at the relationship between the landmark observation and their estimates with the vehicle position measurements and estimates. Ideally good landmark measurements/estimates improve the EKF's ability to estimate vehicle position. If the vehicle knows where a landmark is and it can accurately measure that landmark, it is an indirect measurement of its own position as the measurement is vehicle relative. Figures 3.35, 3.36, and 3.37 are the error RMS graphs for the EKF position estimates in the first scenario. Looking at the first figure, the EKF performs

almost identically well at 1m with landmark observations and without. The runs with observations seem to perform slightly better at the start, but both lines level out at about the same height. Moving on to the second figure with 2m of GPS error, by the end they perform on similar levels, but what is interesting is the early lead present in the runs with measurements has grown. This is an interesting feature which again shows up in figure 3.37 which shows the results with GPS error 3m. The early lead is not as profound - however the run with observations spikes up until 25 seconds, then comes back down. In both figures 3.36 and 3.37 which are the 2m and 3m error runs, the observation run seems to have a downward trend vehicle RMS position error, while the non-observation runs have an upward trend. These trends are more significant in the 3m error run. In all runs there is a peak RMS position error for the observation runs. This peak occurs at the same time the vehicle transitions from strafing to its right back to its left. This is seen often with the EKF during transitions, during which estimates can become more uncertain. It is interesting however that they are so poor as to cause a huge spike in the error RMS. The EKF does a good job estimating the vehicle velocity when it is constant, once the vehicle's velocity transitions, it begins estimating the position with what is not an incorrect velocity. This drastically reduced the EKF accuracy. Typically, the EKF does better during constant motion. It seems that this transition, tied with that it is now estimating to measure landmarks from a poorly estimated position, particularly increases position error. For example, if the EKF continues to estimate movement to the positive x-direction of some magnitude then it will have a more positive x-position when in fact the movement has either slowed down or begun in the negative x-direction then the estimated vehicle position is more incorrect than typical when landmark observations are made causing greater residual for landmark measurements.

Moving on to the second scenario where the vehicle moves towards and away from the wall, figures 3.40, 3.41, 3.42. The results here are different from the first scenario. Here the vehicle measures the same landmarks. This is significant because when the vehicle was strafing, for half of the

motion the vehicle is observing new landmarks. In this scenario all of the landmarks being measured are discovered within the first few seconds of the simulation. For all three runs it can be seen that the tests with observations outperform those without. This is significant. What is happening is that the vehicle is measuring the same landmarks here in this scenario where the previous scenario the vehicle was measuring new landmarks for half of the simulation time.

When looking closely at figures 3.36 and 3.37 which show the 2m and 3m errors for the strafing tests, a downward trend can be seen for the observation runs. This is significant especially when looking at the data from the second scenario. Recalling the second scenario does not measure new landmarks after the first few seconds of simulation time, after the 25 second hump there should be no new landmarks being measuring by the strafing vehicles. There appears to be improvement in estimate accuracy that occurs once no new landmarks are being measured.

4.12 VEHICLE LOITER AND SOURCE LOCALIZATION UPDATE TIME LAG TEST DISCUSSION

For this test we are investigating the time lag for localizing the source. There are a few factors that affect the speed at which the EKF will change an estimate. Here, the vehicle loiters at a set location relative to the radioactive source. The closer the location is to the source, the better the measurements are. As shown, the closer the vehicle is to the source the more accurate the estimate is. For the case where the source is somewhat successfully localized (positions 1 and 2), the closer better measurements localize much faster. Locations 3 and 4, either see little estimate improvement or decreased estimate accuracy. This alone highlights a weakness of the EKF, something seen before, which is that the EKF does poorly with radiation estimates when the vehicle is too distance from the source.

Looking closely at figure 3.43 it can be seen at each second a significant change in the estimate. These changes occur when a radiation measurement is done. As the measurement does not match the estimated measurement - a more drastic change in the estimate occurs. When the vehicle is closer the

measurement values estimated and acquired are much more different, causing the update to occur. This shows that the time lag that occurs with estimating is subject to relative position.

It should also be noted that another major factor in the updating process is how the covariance matrices Q, R, and P are initialized. Looking at equations 2.41 and 2.42 it can be seen that the Kalman Gain Matrix K is a function of Q, R, and P. As the estimated covariance matrices are changes (along with the initial error covariance matrix P) so does the gain corresponding to the same difference in estimated measurement and actual measurement. This has a major effect on how fast the EKF will update its estimates. This means that the time lag seen is also a function of initial guesses about the EKF or measurement errors.

4.13 EKF 3 GENERAL DISCUSSION

The final EKF 3 version performs at about the same level as the first KF used in terms of estimates, however by design it is slightly more useful. It would be a good idea to run an experiment looking at the longer term performance of the algorithm; this is true for other KF applications. For many of the results presented, the flight times are less than 60 seconds. In some cases, it is possible for the (E)KF to warm up, in which case short flight times may not present the full scope of the algorithm's performance. Generally, when the RMS Position Error values reach some asymptotic value, the simulation ends shortly after. This is not the case for all results and a longer term study could prove useful.

Since the EKF 3 uses landmarks as opposed to fixed space (either fixed or non-fixed) it does not have an unnecessary 1000+ computations. Additionally, since the EKF 3 estimates the source activity and location it provides slightly more feedback to the user or vehicle, where before additional processing would need to be done to determine the location of the source. These changes go a long way in combatting the estimation time for the EKF. In the first KF application there were over 2000 states estimated, many of which did not need to be estimated - taking a very long time to go through

the KF process. EKF 3 starts with no landmarks, initially only estimating 12 state variables, and as more landmarks are found, more variables are estimated on an as needed basis.

The results and discussion above demonstrate many of the strengths of the EKF, we will now discuss weaknesses related to the vehicle estimation and measurement process. The vehicle estimation is largely the core of the EKF, without which the map forming and radiation estimates would not be possible. The IMU and gyro measurements are used by the EKF in the transition model. These are very prone to sudden high amounts of error which may cause issues with the EKF, especially in the case where that error is perhaps not appropriately simulated in the Matlab simulation. Additionally, GPS also provides the ability to measure velocity, something that is not utilized here, which may improve on the vehicle estimation process. Since the EKF seemed to estimate the vehicle states well enough early on, some useful design changes were not made in order to progress the project towards the goals of mapping and radiation.

In the application of this EKF, it is not only likely but somewhat intended to be used on the open source Ardupilot firmware. The firmware itself is responsible for estimating a great deal of vehicle states and instrument measurements – all of which (both estimates and measurements) are accessible through the Mavlink communication protocol. This is an important fact to consider; the autopilot that would be utilized for any real world application already does a phenomenal job at vehicle estimation, so a focus on vehicle estimation from an external attached package is not as essential with the autopilot readily available for position estimates.

The landmark observation and estimation process is somewhat difficult. After reading the methods section on the observation process and reviewing the code in appendix A for the landmark upgrading process, it may not be apparent that currently the landmark upgrade system has some flaws. A major flaw is that before an unconfirmed landmark is upgraded to a confirmed one, but after being detected some threshold number of times, its position can drastically change from a single inaccurate

measurement. For most runs shown the threshold strength is 10, meaning the landmark must be measured 10 times. These measurements are subject to vehicle estimates – which can be drastically incorrect making a measurement of landmark A seem to be a measurement of landmark B. The issue of position estimates causing incorrect landmarks to be measured is somewhat unavoidable as perfect vehicle estimation is not possible.

One portion of the landmark model that can be changed is that when an unconfirmed landmark is measured, its position is averaged with the new measurement. It takes 10 measurements for an upgrade to occur. The EKF does nothing with (or to) unconfirmed landmarks. As the unconfirmed landmarks are measured the position measured is averaged with the unconfirmed, even if the measurement was unreliable. An unconfirmed landmark can be measured well (accurately to within less than a tenth of a meter) 9 times, while a single measurement can drastically change the position of the unconfirmed landmark. If the unconfirmed is measured 9 times to be at a position 1m, and a single time to be at a position 1.5m then the unconfirmed landmark will be at 1.25m. The overall landmark results are quite good, and this effect was considered, however the decision was made at the time to not do a weighted average. It may be good to add a weighted average so that a single poor measurement doesn't completely displace the landmark location.

While considering a displaced landmark, or basically a landmark in a position that one really does not exist, there is currently no way in the EKF 3 version to allow for the removal of a landmark. This means that poor landmarks will be stuck in the estimated world forever. As discussed above this can drastically affect the obstacle avoidance and vehicle navigation. It would be good to add a routine based on the proximity sensor's range and estimated vehicle position to develop a routine that determines which landmarks should be measured. This would allow for a way to remove confirmed landmarks, where if they are not measured when they should be, they would eventually be removed.

Since computation time can be an issue, the number of confirmed landmarks needs to be considered. Each landmark has 3 variables associated with its position, that means if there are 100 landmarks the EKF state vector is over 300 in length, as the vehicle continues to measure more landmarks this length will only increase, potentially causing the EKF to take too long to estimate. There are a few approaches to address this issue. The first and most simple solution is to use the threshold distance for landmark measurements and increase it. By increasing the threshold distance the resolution of the estimated world decreases, along with the number of potential landmarks.

Another solution would be to limit the number of landmarks being estimated by some distance to the vehicle. Every landmark can be stored separately and only the landmarks that are within range of the vehicle would be part of the EKF state vector. This could also be taken a step farther and rather than distance, instead only estimate the landmarks that are being measured. Currently only the landmarks being measured are said to be measured by the state-to-measurement matrix H. If this same information is used to limit the state vector and format the covariance matrices appropriately – then the length of the state vector would never be very long. By taking this approach, it would also be natural to address the issue with confirmed landmarks being permanent. Using the information of which landmarks are/should be measured then they can begin to me removed from the confirmed list.

The next topic is on the radiation localization and activity estimation. As seen it performs overall quite well. However, from the EKF 3 the feedback is quite minimal – there is no way to tell whether the estimate is accurate or is there another source of information to use in conjunction with the estimates. That is, from the first and second filters, the radiation estimates were based on measurements. They provided a visual feedback of counts per second at some location. From this information, a direction towards a higher measurement site can be acquired. From the EKF 3 estimates it is potentially difficult to extract that direction. What it comes down to is that one simply does not

know how well one can trust the radiation estimates, especially when source activity is completely unknown.

To handle this, it may be interesting to keep the raw measurements and display them much like the estimates in the first and second filters. This would allow for a direction to be acquired visually from the pilot if for some reason the estimates seem to be failing. On the other hand, it may also be worth changing the radiation design altogether. If instead of estimating source location and activity it instead estimates the gradient of the counts per second, this analysis could provide a direction for the vehicle to fly in rather than a location where the source might be. A redesign as such may prove to work even better - especially in the context of multiple radioactive sources.

4.14 GENERAL SIMULATION IMPROVEMENTS

There are a number of improvements that can be made regarding the Euler Angle assumption and control implementation. The Euler Angle assumption made, where the roll, pitch and heading angles are used in the construction of the DCM matrices to rotate between the inertial and body frame, is incorrect. Through limited experimentation, done after the assumption was made, the roll and pitch angles do not grow large for long periods of time. This means maneuvers are not extreme and are quite short, with roll and pitch values often equal to zero. During a maneuver period these values are at their greatest and allow for large error in both estimates and actual integrated values. It is proper to both calculate and estimate the correct Euler Angles by applying the transformation onto the body-frame rates as seen in equation 2.4. The results presented here are somewhat in question due to this assumption, however for the majority of the flight, since the first Euler Angle used in the rotation (psi) about the 3rd inertial axis is referenced correctly, the assumption is correct for a large portion of the vehicle flight. It primarily effects the legitimacy of the observations made during a pose changing maneuver and how the thrust vectors effect the vehicle flight during that maneuver. Nonetheless, a more proper implementation of a Euler Rotation should be utilized.

There are two very common practices in controls not implemented in the simulation regarding the vehicle controls. The first is gain-scheduling on the computing of the LQR gain matrix. Simply put, computing the LQR gain at every simulation time step is inefficient. Computing the gain at every time step is done because the vehicle behaves differently as the state variables change. As an example, consider the direction the vehicle faces. If the vehicle faces along the inertial x-axis, a pitch down will cause motion along the x-axis. If now the vehicle faces the inertial y-axis, the same pitch down maneuver resulting from the identical control input will instead cause motion along the y-direction. Clearly, there is different behavior as the vehicle's forward facing direction changes. A typical solution to this is called gain-scheduling, where instead of computing the gain every time step, after the variables enter a new region where the behavior is different from the last, a new gain will be computed or applied at the current state, which will hold until the states have entered another operating region. In this application the vehicle heading is likely the scheduling variable to be used; as the vehicle turns into different operating regions the respective gains are applied. The idea of gain-scheduling could potentially apply beyond the controller gains and perhaps could be utilized to limit the computations of the (E)KF.

The second consideration is that typically the error signals sent to the LQR should be "small," but in some cases in our simulation, large error signals resulted (small and large being defined by the degree of linearity of the system), which degraded control of the vehicle movement. This is mostly seen when sending a "desired" waypoint to the LQR where the velocity is zero, much like equation 2.26. This type of error signal is inefficient and could potentially lead to a crashing maneuver. Instead, what should be applied is a more advanced version of the second layer of controls (discussed in methods section) where a continuous maneuver profile is used to ramp up the vehicle safely and efficiently to some cruising speed or velocity. A simple profile with a 0.5 second ramp up and ramp down period with constant speed in the center could be sufficient. A series of error signals would have to be developed

and sent to the LQR that mimic the maneuver profile. These error signals would provide smaller errors than the signals provided currently in the simulation. By using a maneuver profile and gradually sending the LQR error signals matching the profile, it is likely to produce smoother flightpath results with less overshooting of the vehicle position.

4.15 SUMMARY

Each Kalman Filter has been a step in the right direction towards integrating radiation and proximity data into the flight of an unmanned aerial vehicle. No iteration has been without flaws but only after developing them are those flaws apparent and improvements can be made. Coming to the final iteration of the EKF 3 version, the algorithm has been shown to work quite well in a simulated environment. With the overall general design to be that of an onboard computer with access to the flight controller and other used relevant sensors, it is at the stage where the algorithm could potentially undergo actual real world tests. For the case of most published research the first stage is to simulate the environment and test out a general feasibility of the algorithms developed. At this point continuation could go in one of two directions. The first would be to convert the EKF 3 and relevant routines to a lightweight C/C++ to be used by an onboard computer, and conduct real world tests. These tests would likely first explore the vehicle positon and environment estimates before moving on to source estimate. The source estimates would be difficult to tests as large detectable sources are not easily used or accessed.

The second course of direction (which would lead to the first eventually) would be to redesign some of the components of the algorithm, from improved landmark estimation to improved radiation estimation designs as discussed in section 4.13. To go this direction would involve more development on the simulation code particularly along the lines of the EKF estimation process and routines that would need to be used to prepare measurements and data for the EKF. However, this would lead to real world testing.

Overall this work lays a foundation for continuing the project towards real world testing. With the development of the EKF and additional routines, one would likely need to convert the code from Matlab into a different language. Real world testing may also require changes to the measurements and measurement model as the laser range scanner simulated here is unlikely to be used during a real test due to limited funding. A more realistic sensing model would be a few Lidar like range finders pointed in multiple directions around the vehicle. This would still allow obstacle sensing but severely limit the resolution of what can be sensed in a single vehicle pose.

The Matrix-S that was developed will provide an excellent platform to start testing with. It has excellent flight and lift capabilities allowing for the implementation of multiple range and radiation sensors. Additionally, a key component to its design was the decision to the Pixhawk flight controller. The Pixhawk flight controller (FC) uses ArduPilot open source UAV software. This software then uses a Mavlink protocol which would be the means of communication between the FC and the onboard computer implementing the algorithms developed by this work. The FC can provide access to the multiple onboard sensors (IMU, Gyro, and GPS) to be used by the algorithm. In addition to sensor measurements it can also provide its own set of onboard estimates.

Returning to some of the related work mentioned above, the potential to combine some work is also present. For example in Cetin's and Yilmaz's (Cetin and Yilmaz 2016) paper the potential for representing the world in a manner similar to theirs may work well. It of course runs into similar issues with limiting size of the area being estimated but by estimating only a 2-D matrix of values could substantially reduce the time required for the KF estimation process. Additionally, some papers both Cetin's (Cetin and Yilmaz 2016) and Chang's (Chang, Xia et al. 2016) papers both utilize parallel processing. This is an important aspect the computing capabilities and the KF process at first glance appears to be a parallelizable process. By Parallelizing the KF process this may lift some computation time constraints and allow for more states to be allowed. The ability for the KF to estimate quickly

enough for real time use, the ability to speed it up through parallel processing, could be essential to progressing the area of this research.

CHAPTER 5: CONCLUSION

The problem with autonomous drones is their inability to see the environment as humans see it. Many autonomous vehicles can exist safely at high altitudes as the probably of hitting an obstacle is quite now. In low altitude flight in a more populated environment this is not the case. In order to achieve some level of autonomy it is important to develop a model for which the drone can "see" and understand the environment around it. Once this is achieved then drones can truly become autonomous. This research sought to develop an algorithm that using onboard sensors could begin to allow for the drone to understand the surrounding environment. By simulation the vehicle is able to detect environmental obstacles, radiation data, take GPS, IMU, and gyro data. The algorithms developed incorporates this information using an Extended or traditional Kalman Filter. The Kalman Filter is there to estimate the state of the vehicle, the surrounding environment and radiatioactive data as well. For the first two algorithms the radiation estimation was limited to counts per second at specific locations of radiation measurements, however the third EKF algorithm attempts to estimate the source location and activity.

The first KF algorithm developed allowed for a proof of concept through the KF process. It had severe limitations with both computation time and on the amount of space being estimated. The limit of the space being estimated only allowed for flight within a small 10m by 10m by 10m volume. It however performed quite well in terms of the results presented and shows as a potentially viable algorithm if space and time were not a limitation.

The second EKF algorithm proved to be the most difficult and least effective method. Its design was heavily influenced by computation time and removing the limit of the space being estimated. As a result, the transition model does not work well for the EKF process. Results are reasonable during a loiter however once motion occurs the results are very poor making it impossible to produce useable

estimates for making decisions toward vehicle navigation. The second EKF is the least viable method in large part to the transition model and the poor predictions made as a result of it.

The third EKF is largely a reboot of the previous two. It redesigns the spatial environment model by estimating landmark position as opposed to cube occupancy. This change allows the vehicle to travel anywhere in the world and removes the spatial limit imposed by the first KF. The radiation model changed vastly as well, instead it attempts to localize the radioactive source and update a potential source activity. As measurements are taken the EKF checks the predicted measurements based off the estimated values with the actual measurement values and continually updates the source's estimates. Out of the three, the third EKF shows the most promise for real world testing/implementation.

It is the hope this research to push the level of autonomy forward for Unmanned Aerial Vehicles particularly in an unknown lower altitude environment where hazardous conditions exist. By combining radiation elements into, a potential application for UAVs are explored while many other first responder or general environmental sensing applications exists. Applying the algorithms developed, in particular the third, could allow for a means of the vehicle to understand its surrounding environment giving it the means to have true autonomy.

BIBLIOGRAPHY

Cai, C., B. Carter, et al. (2016). "Designing a Radiation Sensing UAV System." 2016 Ieee Systems and Information Engineering Design Symposium (Sieds): 165-169.

Gathering radiation information in the event of a nuclear disaster is limited by the vulnerability of static sensor networks and the safety of human data collectors. A remotely controlled drone that can combine locational and radiological data gives emergency responders a safe and efficient alternative in the event of damaged or destroyed static sensor networks. The goal of this project is to develop a prototype radiation detection and mapping unmanned aerial vehicle system to safely identify irradiated areas in the event of a nuclear emergency. The UAV platform for this project is the EDF-8, a small ducted-fan UAV, developed by AVID LLC. Our team is responsible for developing the hardware necessary to integrate the radiation sensor into the EDF-8 platform. The main tasks in this project include programming microcontrollers to communicate with the radiation sensor as well as EDF-8's on board communication protocols, building housings for both the sensor and circuit boards, building simulation programs for testing, and the manipulation of collected data for presentation in a GIS software program. Upon project completion, AVID will have a prototype system for adapting EDF-8 into a radiological detection system for commercial or governmental use.

Cetin, O. and G. Yilmaz (2016). "Real-time Autonomous UAV Formation Flight with Collision and Obstacle Avoidance in Unknown Environment." Journal of Intelligent & Robotic Systems **84**(1-4): 415-433.

In this paper autonomous formation control for Unmanned Aerial Vehicles (UAVs) has been discussed and a real time solution has been put forward by benefiting General Purpose Graphical Processing Units (GPGPU) accelerated potential field approach while ensuring obstacle and collision avoidance in unknown environment by using real-time sensors. GPGPU accelerated real time formation control for UAVs was designed and the basic model of the approach has been explained in our previous work (Cetin and Yilmaz 2014). As the deficiencies of the previous approach, autonomous real time collision and coordinated obstacle avoidance features in unknown environments are also handled while maintaining formation flight conditions in this work. With these features, improved autonomous formation control approach is discussed as a real time solution. The computation is performed by using Graphical Processing Units (GPUs) as parallel computation architectures by benefiting from Single Instruction Multiple Data (SIMD) type parallel algorithms. Classic binary map conversation, connected component labeling and minimum bounding box algorithms which are commonly used for image processing applications, has been evaluated for real time obstacle detection and avoidance features by developing GPGPU suitable parallel algorithms. Real-time solution has been developed by integrated these parallel algorithms with parallel Artificial Potential Field (APF) computation algorithm. Simulation results are proved that this novel autonomous improved formation control approach is successful and it would be used in real time applications like UAV formation flight missions.

Chang, K., Y. Q. Xia, et al. (2016). "Obstacle avoidance and active disturbance rejection control for a quadrotor." Neurocomputing **190**: 60-69.

In this paper, an active disturbance rejection control (ADRC) system is developed for autonomous quadrotor with obstacle avoidance. In this control system, the controller based on ADRC technique is the main controller. The robust trajectory tracking problem of the quadrotor is solved based on the attitude decoupling control. ADRC attitude decoupling controllers are

used to eliminate the effect of the state coupling and uncertainties caused by internal and external disturbances. This control system also contains an avoidance obstacle approach whose trajectories can be obtained through two kinds of potential field with rotation vectors, quadrotor can choose a smoother trajectory from them to avoid the obstacle. The effectiveness of the proposed approach has been verified in simulations. (C) 2016 Elsevier B.V. All rights reserved.

Christie, G., L. J. Stiltner, et al. (2014). Synchronous Radiation Sensing and 3D Urban Mapping for Improved Source Identification. Multisensor, Multisource Information Fusion: Architectures, Algorithms, and Applications 2014. J. J. Braun. Bellingham, Spie-Int Soc Optical Engineering. **9121**.

The acquisition of synchronous EO imagery and gamma radiation data in aerial overflights of an unmanned aircraft can provide valuable spatial context for radioactive source mapping. Using image-based 3D reconstruction methods, a terrain map can be generated and used to reason about more likely radiation source locations. For instance, vehicles may be likely hiding places for nuclear materials, so a source model with assigned probability is used at the vehicle to reduce the overall uncertainty in position estimation. Environment reconstructions based on EO imagery with a mapped gamma radiation overlay provide intrinsic correlations between the datasets. Using radioactive material dispersion models or point source models, the derived correlations serve to enhance coarse gamma radiation data. The use of autonomous unmanned aircraft provide a valuable tool in acquiring these data as they are capable of accurate and repeatable position control while eliminating exposure danger to the operators. In this experiment, two sources (.084 Ci Ce-137 and .00048 Ci Ba-133) were distributed in a field with varying terrain and a scan was conducted using the Virginia Tech Yamaha RMAX autonomous helicopter equipped with a two-camera imaging system and a NaI scintillation-type spectrometer. Terrain reconstruction was conducted using both structure from motion (SfM) and stereo vision techniques, and radiation data synchronized to the imagery was overlaid.

Dissanayake, M. W. M. G., P. Newman, et al. (2001). "A solution to the simultaneous localization and map building (SLAM) problem." IEEE Transactions on Robotics and Automation **17**(3): 229-241.

The simultaneous localization and map building (SLAM) problem asks if it is possible for an autonomous vehicle to start in an unknown location in an unknown environment and then to incrementally build a map of this environment while simultaneously using this map to compute absolute vehicle location. Starting from estimation-theoretic foundations of this problem, the paper proves that a solution to the SLAM problem is indeed possible. The underlying structure of the SLAM problem is first elucidated. A proof that the estimated map converges monotonically to a relative map with zero uncertainty is then developed. It is then shown that the absolute accuracy of the map and the vehicle location reach a lower bound defined only by the initial vehicle uncertainty. Together, these results show that it is possible for an autonomous vehicle to start in an unknown location in an unknown environment and, using relative observations only, incrementally build a perfect map of the world and to compute simultaneously a bounded estimate of vehicle location. The paper also describes a substantial implementation of the SLAM algorithm on a vehicle operating in an outdoor environment using millimeter-wave radar to provide relative map observations. This implementation is used to demonstrate how some key issues such as map management and data association can be handled in a practical environment. The results obtained are cross-compared with absolute locations of the map landmarks obtained by surveying. In conclusion, the paper discusses a number of key issues raised by the solution to the SLAM problem including suboptimal map-building algorithms and map management

Haykin, S. (2001). Kalman Filtering and Neural Networks, Wiley.

Hokuyo Automatic Co., L. (2009). "Hokuyo." 2017, from https://www.hokuyo-aut.jp/index.html.

Kim, P. and L. Huh (2011). Kalman Filter for Beginners: With MATLAB Examples, Createspace Independent Pub.

Kleeman, L. Understanding and Applying Kalman Filtering, Department of Electrical and Computer Systems Engineering Monash University, Clayton.

Krane, K. S. (1987). Introductory Nuclear Physics, Wiley.

MathWorks (2017). "Matlab Product Documentation." from https://www.mathworks.com/help/.

Sauer, T. (2012). Numerical Analysis, Pearson Education.

Welch, G. and G. Bishop (2007). An Introduction to the Kalman Filter, University of North Carolina at Chapel Hill.

APPENDIX A: EKF 3 CODE

```
function main ()
    %%This function simulates the flight of a quadcopter in
a 3-d
    %%environment.  During flight sensors are used to
measure orientation
    %%of the vehicle, and detect objects in the
environment.  Version 2
    %%picks up where the c++ version 1 picks off.  Matlab
is used here and
    %%replaces all the libraries.  Due to the lqr()
function not being
    %%available in c/c++ alongside other required libraries
it was decided
    %%to use matlab so that the gain
    %%matrix can be computed in real time for the drones
specific state.
    clear all
    close all

    %setup global variables
    disp('Setting up global variables');
    global dt;
    global Uo;
    global desiredX;
    dt = .01;
    Uo = [11.025 11.025 11.025 11.025 0]';
    desiredX = [20 50 10 0 0 pi/2 0 0 0 0 0]';
    global accumErr;
    accumErr = [0 0 0]';
    global range_std;
    global gps_std;
    global rad_std;
    global ang_std;
    range_std = .5;
    gps_std = .1;
    rad_std = 3;
    ang_std = pi/10;

    global testVar;
    testVar = 0;
```

```
    %world1 = build_world(lx,ly,lz,1);
    %[xplot yplot zplot] = plot_world(world1, lx, ly, lz,
1,0);

    %Define Struct for observations
    disp('Setting up Observations Struct')
    global Observations
    Observations = struct('X',{},'Xact', {}, 't', {},
'Obs', {}, 'Err', {});
    global numObs;
    numObs = 0;



    %set up radiation variables
    disp('Setting up Rad variables');
    global RadObs;
    global Counts;
    global Sources;
    global numSources;
    global numRadObs
    numRadObs = 0;
    numSources = 0;
    Sources = struct('Pos', {}, 'Activity', {});
    Counts = 0;
    RadObs = struct('Counts', {}, 'Pos', {}, 't', {});


%   add in radioactive sources
%   add_source([9 9 3]',1000);
%     add_source([1 9 3]',2000); %this source for the
generic outdoor
%      add_source([10 10 6]',4000); %urban source
    add_source([10 10 2]',4000);
%     gpserr = 1:10;
%     Ezerr = 1:10;
%     k = 1;
%     time = zeros(1,500);
%     pf  = zeros(1,500);
%     rms  = zeros(1,500);
%     Ezerr  = zeros(1,500);
%     gpserr = zeros(1,500);
%     for m = 1:5
```

```
%         for i = 1:10
%             for j = 1:10
%                    disp('Run number:');
%                    disp(k);
%                 [tmp1 tmp2 tmp3] =
avoidwall_withIMU(i,j);
%                 time(k) = tmp1;
%                 pf(k) = tmp2;
%                 rms(k)= tmp3;
%                 Ezerr(k) = j;
%                 gpserr(k) = i;
%                 k = k + 1;
%                 save('1-12-17-
AvoidWallIMUFixxedDetectEnv2');
%             end
%         end
%     end
%     leash_motion7_withIMU();
%     avoidwall_withIMU
%     world_output();
%      motion_withIMU_EKF();
%          motion_SimpleWallAvoidance(.5,.02,1)
%          motion_SimpleWallAvoidance(.5,.02,2)
%          motion_SimpleWallAvoidance(.5,.02,3)
%          motion_SimpleWallAvoidance(.5,.02,4)
%          motion_SimpleWallAvoidance(.5,.02,5)
%          motion_SimpleWallAvoidance(.5,.02,6)
%          motion_SimpleWallAvoidance(.5,.02,7)
%          motion_SimpleWallAvoidance(.5,.02,8)
%          motion_SimpleWallAvoidance(.5,.02,9)
%          motion_SimpleWallAvoidance(.5,.02,10)
%          motion_SimpleWallAvoidance(1,.02,11)
%          motion_SimpleWallAvoidance(1,.02,12)
%          motion_SimpleWallAvoidance(1,.02,13)
%          motion_SimpleWallAvoidance(1,.02,14)
%          motion_SimpleWallAvoidance(1,.02,15)
%          motion_SimpleWallAvoidance(1,.02,16)
%          motion_SimpleWallAvoidance(1,.02,17)
%          motion_SimpleWallAvoidance(1,.02,18)
%          motion_SimpleWallAvoidance(1,.02,19)
%          motion_SimpleWallAvoidance(1,.02,20)
%          motion_SimpleWallAvoidance(1.5,.02,21)
%          motion_SimpleWallAvoidance(1.5,.02,22)
%          motion_SimpleWallAvoidance(1.5,.02,23)
```

```
%           motion_SimpleWallAvoidance(1.5,.02,24)
%           motion_SimpleWallAvoidance(1.5,.02,25)
%           motion_SimpleWallAvoidance(1.5,.02,26)
%           motion_SimpleWallAvoidance(1.5,.02,27)
%           motion_SimpleWallAvoidance(1.5,.02,28)
%           motion_SimpleWallAvoidance(1.5,.02,29)
%           motion_SimpleWallAvoidance(1.5,.02,30)
%           motion_SimpleWallAvoidance(2,.02,31)
%           motion_SimpleWallAvoidance(2,.02,32)
%           motion_SimpleWallAvoidance(2,.02,33)
%           motion_SimpleWallAvoidance(2,.02,34)
%           motion_SimpleWallAvoidance(2,.02,35)
%           motion_SimpleWallAvoidance(2,.02,36)
%           motion_SimpleWallAvoidance(2,.02,37)
%           motion_SimpleWallAvoidance(2,.02,38)
%           motion_SimpleWallAvoidance(2,.02,39)
%           motion_SimpleWallAvoidance(2,.02,40)
%           motion_SimpleWallAvoidance(3,.02,41)
%           motion_SimpleWallAvoidance(3,.02,42)
%           motion_SimpleWallAvoidance(3,.02,43)
%           motion_SimpleWallAvoidance(3,.02,44)
%           motion_SimpleWallAvoidance(3,.02,45)
%           motion_SimpleWallAvoidance(3,.02,46)
%           motion_SimpleWallAvoidance(3,.02,47)
%           motion_SimpleWallAvoidance(3,.02,48)
%           motion_SimpleWallAvoidance(3,.02,49)
%           motion_SimpleWallAvoidance(3,.02,50)
%           motion_SimpleWallAvoidance(4,.02,51)
%           motion_SimpleWallAvoidance(4,.02,52)
%           motion_SimpleWallAvoidance(4,.02,53)
%           motion_SimpleWallAvoidance(4,.02,54)
%           motion_SimpleWallAvoidance(4,.02,55)
%           motion_SimpleWallAvoidance(4,.02,56)
%           motion_SimpleWallAvoidance(4,.02,57)
%           motion_SimpleWallAvoidance(4,.02,58)
%           motion_SimpleWallAvoidance(4,.02,59)
%           motion_SimpleWallAvoidance(4,.02,60)
%           motion_SimpleWallAvoidance(5,.02,61)
%           motion_SimpleWallAvoidance(5,.02,62)
%           motion_SimpleWallAvoidance(5,.02,63)
%           motion_SimpleWallAvoidance(5,.02,64)
%           motion_SimpleWallAvoidance(5,.02,65)
%           motion_SimpleWallAvoidance(5,.02,66)
%           motion_SimpleWallAvoidance(5,.02,67)
```

```
%           motion_SimpleWallAvoidance(5,.02,68)
%           motion_SimpleWallAvoidance(5,.02,69)
%           motion_SimpleWallAvoidance(5,.02,70)

%          motion_SimpleWallAvoidance(1,.02,2)
%          motion_SimpleWallAvoidance(1.5,.02,3)
%          motion_SimpleWallAvoidance(2,.02,4)
%          motion_SimpleWallAvoidance(2.5,.02,5)
%          motion_SimpleWallAvoidance(3,.02,6)
%          motion_SimpleWallAvoidance(4,.02,7)
%          motion_SimpleWallAvoidance(5,.02,8)
%          motion_SimpleWallAvoidance(.5,.05,9)
%          motion_SimpleWallAvoidance(1,.05,10)
%          motion_SimpleWallAvoidance(1.5,.05,11)
%          motion_SimpleWallAvoidance(2,.05,12)
%          motion_SimpleWallAvoidance(2.5,.05,13)
%          motion_SimpleWallAvoidance(3,.05,14)
%          motion_SimpleWallAvoidance(4,.05,15)
%          motion_SimpleWallAvoidance(5,.05,16)
%          motion_SimpleWallAvoidance(.5,.1,17)
%          motion_SimpleWallAvoidance(1,.1,18)
%          motion_SimpleWallAvoidance(1.5,.1,19)
%          motion_SimpleWallAvoidance(2,.1,20)
%          motion_SimpleWallAvoidance(2.5,.1,21)
%          motion_SimpleWallAvoidance(3,.1,22)
%          motion_SimpleWallAvoidance(4,.1,23)
%          motion_SimpleWallAvoidance(5,.1,24)

%        motion_withIMU_EKF_urban();



%       motion_RadCovTest([10 10 6 4000], [1 1],1,1)
%       motion_RadCovTest([10 10 6 4000], [1 1],1,2)
%       motion_RadCovTest([10 10 6 4000], [1 1],1,3)
%       motion_RadCovTest([10 10 6 4000], [1 1],1,4)
%       motion_RadCovTest([10 10 6 4000], [1 1],1,5)
%
%       motion_RadCovTest([10 10 6 4000], [1 1],2,6)
%       motion_RadCovTest([10 10 6 4000], [1 1],2,7)
%       motion_RadCovTest([10 10 6 4000], [1 1],2,8)
%       motion_RadCovTest([10 10 6 4000], [1 1],2,9)
%       motion_RadCovTest([10 10 6 4000], [1 1],2,10)
%
```

```
%       motion_RadCovTest([10 10 6 4000], [1 1],3,11)
%       motion_RadCovTest([10 10 6 4000], [1 1],3,12)
%       motion_RadCovTest([10 10 6 4000], [1 1],3,13)
%       motion_RadCovTest([10 10 6 4000], [1 1],3,14)
%       motion_RadCovTest([10 10 6 4000], [1 1],3,15)
%
%       motion_RadCovTest([5 5 6 4000], [5 1],1,16)
%       motion_RadCovTest([5 5 6 4000], [5 1],1,17)
%       motion_RadCovTest([5 5 6 4000], [5 1],1,18)
%       motion_RadCovTest([5 5 6 4000], [5 1],1,19)
%       motion_RadCovTest([5 5 6 4000], [5 1],1,20)
%
%       motion_RadCovTest([5 5 6 4000], [5 1],2,21)
%       motion_RadCovTest([5 5 6 4000], [5 1],2,22)
%       motion_RadCovTest([5 5 6 4000], [5 1],2,23)
%       motion_RadCovTest([5 5 6 4000], [5 1],2,24)
%       motion_RadCovTest([5 5 6 4000], [5 1],2,25)
%
%       motion_RadCovTest([5 5 6 4000], [5 1],3,26)
%       motion_RadCovTest([5 5 6 4000], [5 1],3,27)
%       motion_RadCovTest([5 5 6 4000], [5 1],3,28)
%       motion_RadCovTest([5 5 6 4000], [5 1],3,29)
%       motion_RadCovTest([5 5 6 4000], [5 1],3,30)
%
%       motion_RadCovTest([7 7 4 4000], [3 1],1,31)
%       motion_RadCovTest([7 7 4 4000], [3 1],1,32)
%       motion_RadCovTest([7 7 4 4000], [3 1],1,33)
%       motion_RadCovTest([7 7 4 4000], [3 1],1,34)
%       motion_RadCovTest([7 7 4 4000], [3 1],1,35)
%
%       motion_RadCovTest([7 7 4 4000], [3 1],2,36)
%       motion_RadCovTest([7 7 4 4000], [3 1],2,37)
%       motion_RadCovTest([7 7 4 4000], [3 1],2,38)
%       motion_RadCovTest([7 7 4 4000], [3 1],2,39)
%       motion_RadCovTest([7 7 4 4000], [3 1],2,40)
%
%       motion_RadCovTest([7 7 4 4000], [3 1],3,41)
%       motion_RadCovTest([7 7 4 4000], [3 1],3,42)
%       motion_RadCovTest([7 7 4 4000], [3 1],3,43)
%       motion_RadCovTest([7 7 4 4000], [3 1],3,44)
%       motion_RadCovTest([7 7 4 4000], [3 1],3,45)


        motion_ObsCovTest(0,1,1)
```

```
        motion_ObsCovTest(0,1,2)
        motion_ObsCovTest(0,1,3)
        motion_ObsCovTest(0,1,4)
        motion_ObsCovTest(0,1,5)
        motion_ObsCovTest(1,1,6)
        motion_ObsCovTest(1,1,7)
        motion_ObsCovTest(1,1,8)
        motion_ObsCovTest(1,1,9)
        motion_ObsCovTest(1,1,10)
% %
        motion_ObsCovTest(0,2,11)
        motion_ObsCovTest(0,2,12)
        motion_ObsCovTest(0,2,13)
        motion_ObsCovTest(0,2,14)
        motion_ObsCovTest(0,2,15)
        motion_ObsCovTest(1,2,16)
        motion_ObsCovTest(1,2,17)
       motion_ObsCovTest(1,2,18)
       motion_ObsCovTest(1,2,19)
       motion_ObsCovTest(1,2,20)

        motion_ObsCovTest(0,3,21)
        motion_ObsCovTest(0,3,22)
        motion_ObsCovTest(0,3,23)
        motion_ObsCovTest(0,3,24)
        motion_ObsCovTest(0,3,25)
        motion_ObsCovTest(1,3,26)
        motion_ObsCovTest(1,3,27)
        motion_ObsCovTest(1,3,28)
        motion_ObsCovTest(1,3,29)
        motion_ObsCovTest(1,3,30)

%     motion_LoiterSourceDetection([1 1],1000,[10 10 4],1)
%     motion_LoiterSourceDetection([1 1],1000,[10 10 4],2)
%     motion_LoiterSourceDetection([1 1],1000,[10 10 4],3)
%     motion_LoiterSourceDetection([1 1],1000,[10 10 4],4)
%     motion_LoiterSourceDetection([1 1],1000,[10 10 4],5)

%     motion_LoiterSourceDetection([1 1],1000,[7.5 7.5
5],6)
%     motion_LoiterSourceDetection([1 1],1000,[7.5 7.5
5],7)
%     motion_LoiterSourceDetection([1 1],1000,[7.5 7.5
5],8)
```

```
%     motion_LoiterSourceDetection([1 1],1000,[7.5 7.5
5],9)
%     motion_LoiterSourceDetection([1 1],1000,[7.5 7.5
5],10)
%
%
%     motion_LoiterSourceDetection([1 1],1000,[5 5 5],11)
%     motion_LoiterSourceDetection([1 1],1000,[5 5 5],12)
%     motion_LoiterSourceDetection([1 1],1000,[5 5 5],13)
%     motion_LoiterSourceDetection([1 1],1000,[5 5 5],14)
%     motion_LoiterSourceDetection([1 1],1000,[5 5 5],15)


    keyboard;
end

function motion_withIMU_EKF()
    global Sources
    global RadObs;
    global numRadObs
    global dt;

    global desiredX;
    global SensorStatus
    global decision
    global IMU

    IMU = [0 0 0 0 0 0];
    SensorStatus = [0 0 0 0 0 0];

    global gotRadMeasurements;

    gotRadMeasurements = 0;

    %setup world related variables
    disp('Building World');
    ppm = 5;
    lx = 100;
    ly = lx;
    lz = ly;

    world = build_world_simple2(20,20,20,ppm);
    observed_world = build_world_empty(20,20,20,.5);
```

```
    world1 = build_world_simple2(20,20,20,5);
    [xplot, yplot, zplot] = plot_world(world1, 20, 20, 20,
5,0);
    figure
    plot3(xplot,yplot,zplot,'b.')
    axis([0 20 0 20 0 20])
%     keyboard
    %set up space for occupancy and radiation cubes
    Occ = cubespace(10,10,10,2);
    Rad = cubespace(10,10,10,2);
    Ncubes =
length(Occ(:,1,1))*length(Occ(1,:,1))*length(Occ(1,1,:));

    States = struct('X',{});
    States(1).X = [10 8 8 0 0 pi/2 0 0 2 0 0]';
    waypoints = struct('X',{});
    waypoints(1).X = [5 15 2]';
      waypoints(2).X = [15 3 5]';
      waypoints(3).X = [15 5 7]';
      waypoints(4).X = [12 5 3]';
      waypoints(5).X = [5 12 5]';
%       waypoints(6).X = [2 3 5]';
%      waypoints(7).X = [9 4 10]';
%      waypoints(8).X = [1 9 5]';
%      waypoints(9).X = [5 9 5]';
%      waypoints(10).X = [1 9 5]';
%      waypoints(11).X = [1 6 5]';
%      waypoints(12).X = [1 15 5]';
%     waypoints(13).X = [2 7 5]';
%     waypoints(14).X = [9 7 5]';
%     waypoints(15).X = [9 8 5]';
%     waypoints(16).X = [2 8 5]';
%     waypoints(17).X = [2 9 5]';
%     waypoints(18).X = [9 9 5]';
%     waypoints(1).X = [2 3 5]';

    Vdes = 2;
    X = [5 9 2 0 0 pi/2 0 0 0 0 0]';
    tsteps = 20;
    t = 0;
    dt = .01;
    deltaT = tsteps*dt;
    j = 1;
    C = 2*pi*5;
```

```
T = C/Vdes;
f = 1/T;
w = 2*pi*f;
States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 5 0 0 0]';

%States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 .5 0 0]';
Nstates = length(X) + 2*Ncubes;

%Build some initial errors for Ex
VXYZerr = .1;
VRPYerr = .1;
sigmaVx = .1;
sigmaVy = .1;
sigmaVz = .1;
sigmaYdot = .1;
sigmaX = deltaT*sigmaVx;
sigmaY = deltaT*sigmaVx;
sigmaZ = deltaT*sigmaVx;
sigmaRoll = .1;
sigmaPitch = .1;
sigmaYaw = deltaT*sigmaYdot;
ObjErr1 = .5;
RadErr1 = [.25 100];


%errors associated with Ez
GPSerr = 1.5;
CompassErr = .1;
ObjErr2 = 1;
RadErr = 5;
Xerr = 0;
Yerr = 1;
Zerr = 1;
YAWerr = 0;
ObjMeasErr = .001;
RadMeasErr = .5;

tog =1;
lastx = X(1);
lasty = X(2);
lastz = X(3);

deltaT = tsteps*dt;
I = eye(Nstates);
```

```
    Nstates = 9 + 2*Ncubes;
    I = eye(Nstates);
    [sensX, sensY, sensZ, Nbeams] =
build_sensor_array(5*pi/6, pi/3,5);
    Nmeasurements = 6 + Ncubes + 1;
    ObjIndices = 0;
    RadIndex = 0;

%     Hk = build_imuHk(Nstates,ObjIndices,
Ncubes,Nmeasurements, RadIndex);
%     Ex =
build_imuEx(Nstates,sigmaVx,sigmaVy,sigmaVz,sigmaYdot,Objer
r,Raderr,deltaT);
%     Ez = build_imuEz(Nmeasurements,Nbeams ,[Xerr
YAWerr],ObjMeasErr, RadMeasErr);
     Xk(:,1) = zeros(12,1);
    Xk(1:9,1) = [X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8)
X(9)];
    Xk(10:13,1) = [10 10 2 2000];
%     load('wallandtree.mat')
% xtemp = [5 0 4]';
% xdestemp = [5 9 4]';
% Vdir = (xdestemp - xtemp)/norm(xdestemp - xtemp);
%     scatter3(xobs-.5,yobs-.5,zobs-.5,obs_str,'b');

%     keyboard;
    P = zeros(Nstates,Nstates);
%     plot3(Xk(1,1:end-1),Xk(2,1:end-1),Xk(3,1:end-1),'b')
    Zk = zeros(Nmeasurements,1);
    Zk(1:4,1) = [Xk(1,1) Xk(2,1) Xk(3,1) Xk(6,1)]';
    tog = 0;
    HeadingState = States(1).X - X;
    Grad = 0;
    %Just initial flight/sensing test routine
    disp('Begin sim');

    figure;


    Pi = [0 0 0]';
    lpi = 0;
    lq = 0;
```

```
    qi = [0 0 0]';
    pf = [0 0 0]';
    qstr = 0;
    ti = 0;
    firstPass = 1;
    landmarkexists = 0;


   NumberOfObservations = detect_env3_imu_cleaned(world, X,
[20 20 20], ppm,t,sensX, sensY, sensZ,Nbeams,Ncubes,Xk);
   [Pi qi pf ti,which_pi_measured lpi lq landmarkexists
qstr Zobj] = Parse_Observations(Pi, lpi, qi, lq, qstr, ti,
t, NumberOfObservations,firstPass,landmarkexists);
   Xk = fix_state_vector(Xk, lpi, Pi);
   P = build_Po(Xk);
   Ex = build_Ex(lpi, VXYZerr, VRPYerr, ObjErr1, RadErr1,
deltaT);
    firstPass = 0;
    for i = 1:numel(waypoints)     %Navigate to each point

        decision = 0;
        desiredX = States(1).X;     %set desired state
        finalPos = States(1).X(1:3);
        disp('Detecting/Integrating');

        Vdir = (waypoints(i).X(1:3) -
X(1:3))/norm(waypoints(i).X(1:3) - X(1:3));

        States(1).X = [X(1) X(2) X(3) 0 0 pi/2 Vdes*Vdir(1)
Vdes*Vdir(2) Vdes*Vdir(3) 0 0]';
        currentWay = waypoints(i).X(1:3);

        while (norm(X(1:3)-waypoints(i).X(1:3))>=.5)
%Once within some distance stop move to next point

%             while (t<=30)


%             add_gps_err();
%function to add random walk aspect


            Xkest = TransitionModel(Xk,IMU',deltaT);
            [Ak, ~] = get_Ak_jacobian2(Xk,IMU');
```

```
            P = Ak*P*(Ak')+Ex;



            A = RK4(@EOM2,X,t,dt,tsteps,0);
%integrate motion for 20 time steps
            detect_rad(X(1:3),deltaT);

            NumberOfObservations =
detect_env3_imu_cleaned(world, X, [20 20 20], ppm,t,sensX,
sensY, sensZ,Nbeams,Ncubes,Xk);
            [Pi qi pf ti,which_pi_measured lpi lq
landmarkexists qstr Zobj] = Parse_Observations(Pi, lpi, qi,
lq, qstr, ti, t,
NumberOfObservations,firstPass,landmarkexists);
            Xkest = fix_state_vector(Xkest, lpi, Pi);
            P = Fix_Pk(P,Xkest);
            Ex = build_Ex(lpi, VXYZerr, VRPYerr, ObjErr1,
RadErr1, deltaT);


            if ((mod(t,1) < .00001) || (abs(mod(t,1)-1) <
.0001))       %every 1 second store radiation data
                if (t ~= 0)
                    gotRadMeasurements = 1;
                    disp('Recording Counts');
                    record_counts(X(1:3),t);
%record radiation data
                    disp(X(1:3)');
                    disp(t);
                    disp(Xk(length(Xk)-3:length(Xk)-1));
                    disp (norm(Xk(1:3)-X(1:3)));
                    disp (RadObs(numRadObs).Counts)
                    rSquared = (Xk(length(Xk)-3) -
Xk(1))^2 + (Xk(length(Xk)-2) - Xk(2))^2 + (Xk(length(Xk)-1)
- Xk(3))^2;
                    disp(Xk(end)/rSquared);
                    disp(P(end,end));
                end
%                   keyboard;
%                   RadIndex =
find_measurement_index2_imu(Xkest(:,j), [0 0 0]');
```

```
%                   ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,RadObs(end).Counts);
%                       RadIndex = 0;
%                   ZMeas =
handle_Measurements(X,Nmeasurements,[1 1 1 .1],tmpMeas,20);
%               keyboard;
%               else
%                   RadIndex = 0;
%                   ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,0);
            end

            Zest = ObservationModel(pf, which_pi_measured,
Xkest);
            Ez = build_Ez(length(Zest), GPSerr, CompassErr,
ObjErr2, RadErr);
            H =
get_Hk_jacobian(Xkest,pf,which_pi_measured);

            Z = GetMeasurement(X, [GPSerr CompassErr],
Zobj, length(Zest),gotRadMeasurements);

            K = P*transpose(H)*(H*P*transpose(H) + Ez)^(-
1);
            Xk = Xkest + K*(Z-Zest);
            I = eye(length(P));
            P = (I - K*H)*P;

%             waypoints(i).X(1:2) = Xk(length(Xk)-
3:length(Xk)-2);
%             currentWay = waypoints(i).X(1:3);
            gotRadMeasurements = 0;


            t = t + tsteps*dt;
%store time
            X = A(end,2:12)';                       %store
position
            x1(j) = X(1);
            y1(j) = X(2);
            z1(j) = X(3);
            roll(j) = X(4);
```

```
            pitch(j) = X(5);
            yaw(j) = X(6);
            Vx(j) = X(7);
            Vy(j) = X(8);
            Vz(j) = X(9);
            time(j) = t;
            x2(j) = Xk(1);
            y2(j) = Xk(2);
            z2(j) = Xk(3);
            radx(j) = Xk(length(Xk)-3);
            rady(j) = Xk(length(Xk)-2);
            radz(j) = Xk(length(Xk)-1);
            activity(j) = Xk(end);
            gpsX(j) = Z(1);
            gpsY(j) = Z(2);
            gpsZ(j) = Z(3);
            yawEst(j) = Xk(6);
            j = j+1;



%               clf
%                plot3(0,0,0,'r')
%               hold on
%               axis([-5 5 -5 5 -5 5])


%              hold off
%              refresh
%              pause(.1)
%



              Vdir = (currentWay - X(1:3))/norm(currentWay
- X(1:3));
              heading = atan2(waypoints(i).X(2) -Xk(2),
waypoints(i).X(1) -Xk(1));
               [tog, xcol] = check_traj2(X,currentWay,
Vdir, Pi, lpi);
              if (tog == 1)
                  disp('Danger');
```

```
                    currentWay =
get_new_way2(xcol,X,currentWay,Vdir, Pi, lpi);
%                   currentWay =
get_new_way(xcol,X,currentWay,Vdir, xobs, yobs, zobs);
%                   tog = 0;
                else
                    currentWay = waypoints(i).X(1:3);
                end

               Vdir = (currentWay - X(1:3))/norm(currentWay -
X(1:3));
               if (norm(X(1:3)-waypoints(i).X(1:3))>=.5)
                   Vdes = .5;
               else
                   Vdes = 5;
               end

                States(1).X = [X(1) X(2) X(3) 0 0 heading
Vdes*Vdir(1) Vdes*Vdir(2) Vdes*Vdir(3) 0 0]';
%               States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 0
0 0]';
               %States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 .5
0 0]';
              desiredX = States(1).X;
%               ErrVec = [(x1' -Xk(1,:)') (y1' -Xk(2,:)')
(z1' -Xk(3,:)')];
%               for p = 1:length(x1)
%                   ErrNorm(p) = norm(ErrVec(p,:));
%               end
%               rms =
sqrt(sum(ErrNorm.*ErrNorm)/length(ErrNorm));
        end

    end

    ErrVec = [(x1' -Xk(1,:)') (y1' -Xk(2,:)') (z1' -
Xk(3,:)')];
    for i = 1:length(x1)
        ErrNorm(i) = norm(ErrVec(i,:));
    end
    rms = sqrt(sum(ErrNorm.*ErrNorm)/length(ErrNorm));
    EstErrX = x1-x2;
    EstErrY = y1-y2;
    EstErrZ = z1-z2;
```

```
    GPSErrX = x1 - gpsX;
    GPSErrY = y1 - gpsY;
    GPSErrZ = z1 - gpsZ;
    RadErrX = Sources(1).Pos(1) - radx;
    RadErrY = Sources(1).Pos(2) - rady;
    RadErrZ = Sources(1).Pos(3) - radz;
    DistFromSourceX = Sources(1).Pos(1) - x1;
    DistFromSourceY = Sources(1).Pos(2) - y1;
    DistFromSourceZ = Sources(1).Pos(3) - z1;

    EstErr = sqrt(EstErrX.^2 + EstErrY.^2 + EstErrZ.^2);
    GPSErr = sqrt(GPSErrX.^2 + GPSErrY.^2 + GPSErrZ.^2);
    RadErr = sqrt(RadErrX.^2 + RadErrY.^2 + RadErrZ.^2);
    DistFromSource = sqrt(DistFromSourceX.^2 +
DistFromSourceY.^2 +  DistFromSourceZ.^2);
    for i  =  1:length(EstErr)
        rmsEst(i) = sqrt(sum(EstErr(1:i).^2)/i);
        rmsGPS(i) = sqrt(sum(GPSErr(1:i).^2)/i);
        rmsRad(i) = sqrt(sum(RadErr(1:i).^2)/i);

    end

    [Xveh, Obj, Rad] = break_state_vector(Xk);



    figure;     %%Plot trajectory results with real world
imposed
    plot3(x1,y1,z1,'r');
    title('Vehicle Trajectory w/ Real World');
    xlabel('X');
    ylabel('Y');
    zlabel('Z');
    hold on
    axis([0 20 0 20 0 20]);
    plot3(xplot,yplot,zplot,'b.')
    hold off

    figure
    plot3(Obj(1,:), Obj(2,:), Obj(3,:),'b.')
    hold on
    plot3(x1,y1,z1,'r');
    title('Vehicle Trajectory w/ Estimated World');
    xlabel('X');
```

```
    ylabel('Y');
    zlabel('Z');
    axis([0 20 0 20 0 20]);
    hold off


    figure
    plot(time,rmsEst,'r--');
    hold on
    title('Estimate and GPS Measurement RMS');
    xlabel('Time (s)');
    ylabel('RMS (meters)');
    plot(time,rmsGPS,'b--');
    legend('Estimated Position RMS', 'GPS Measurement RMS')
    hold off

    figure
    plot(time, RadErr);
    title('Radiation Estimation Error Vs. Time (w/ Relative
Distance Overlay')
    xlabel('Time (s)');
    ylabel('Rad. Err/Distance from Source (meters)');
    hold on
    plot(time, DistFromSource,'r--')
    legend('Radiation Estimate Difference', 'Distance from
Source');
    hold off


    keyboard;

end

function
motion_SimpleWallAvoidance(GPSERROR,XYZERROR,RunNumber)
    global Sources
    global RadObs;
    global numRadObs
    global dt;

    global desiredX;
    global SensorStatus
    global decision
    global IMU
```

```
    IMU = [0 0 0 0 0 0];
    SensorStatus = [0 0 0 0 0 0];

    global gotRadMeasurements;

    gotRadMeasurements = 0;

    wall_collision = 0;

    %setup world related variables
    disp('Building World');
    ppm = 5;
    lx = 100;
    ly = lx;
    lz = ly;

    world = build_world_wall(20,20,20,ppm);
    observed_world = build_world_empty(20,20,20,.5);
    world1 = build_world_wall(20,20,20,5);
    [xplot, yplot, zplot] = plot_world(world1, 20, 20, 20,
5,0);
%     figure
%     plot3(xplot,yplot,zplot,'b.')
%     axis([0 20 0 20 0 20])
%      keyboard
    %set up space for occupancy and radiation cubes
    Occ = cubespace(10,10,10,2);
    Rad = cubespace(10,10,10,2);
    Ncubes =
length(Occ(:,1,1))*length(Occ(1,:,1))*length(Occ(1,1,:));

    States = struct('X',{});
    States(1).X = [10 8 8 0 0 pi/2 0 0 2 0 0]';
    waypoints = struct('X',{});
    waypoints(1).X = [10 15 3]';
%       waypoints(2).X = [15 3 5]';
%       waypoints(3).X = [15 5 7]';
%       waypoints(4).X = [12 5 3]';
%       waypoints(5).X = [5 12 5]';
%       waypoints(6).X = [2 3 5]';
%      waypoints(7).X = [9 4 10]';
%      waypoints(8).X = [1 9 5]';
%      waypoints(9).X = [5 9 5]';
```

```
%       waypoints(10).X = [1 9 5]';
%       waypoints(11).X = [1 6 5]';
%       waypoints(12).X = [1 15 5]';
%      waypoints(13).X = [2 7 5]';
%      waypoints(14).X = [9 7 5]';
%      waypoints(15).X = [9 8 5]';
%      waypoints(16).X = [2 8 5]';
%      waypoints(17).X = [2 9 5]';
%      waypoints(18).X = [9 9 5]';
%      waypoints(1).X = [2 3 5]';

    Vdes = 2;
    X = [10 3 3 0 0 pi/2 0 0 0 0 0]';
    tsteps = 20;
    t = 0;
    dt = .01;
    deltaT = tsteps*dt;
    j = 1;
    C = 2*pi*5;
    T = C/Vdes;
    f = 1/T;
    w = 2*pi*f;
    States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 5 0 0 0]';

    %States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 .5 0 0]';
    Nstates = length(X) + 2*Ncubes;

    %Build some initial errors for Ex
    VXYZerr = .1;
    VRPYerr = .1;
    sigmaVx = .1;
    sigmaVy = .1;
    sigmaVz = .1;
    sigmaYdot = .1;
    sigmaX = deltaT*sigmaVx;
    sigmaY = deltaT*sigmaVx;
    sigmaZ = deltaT*sigmaVx;
    sigmaRoll = .1;
    sigmaPitch = .1;
    sigmaYaw = deltaT*sigmaYdot;
    ObjErr1 = .5;
    RadErr1 = [.25 100];
```

```
    %errors associated with Ez
    GPSerr = GPSERROR;
    CompassErr = .1;
    ObjErr2 = 1;
    RadErr = 5;
    Xerr = 0;
    Yerr = 1;
    Zerr = 1;
    YAWerr = 0;
    ObjMeasErr = .001;
    RadMeasErr = .5;

    tog =1;
    lastx = X(1);
    lasty = X(2);
    lastz = X(3);

    deltaT = tsteps*dt;
    I = eye(Nstates);


    Nstates = 9 + 2*Ncubes;
    I = eye(Nstates);
    [sensX, sensY, sensZ, Nbeams] =
build_sensor_array(5*pi/6, pi/3,5);
    Nmeasurements = 6 + Ncubes + 1;
    ObjIndices = 0;
    RadIndex = 0;

%     Hk = build_imuHk(Nstates,ObjIndices,
Ncubes,Nmeasurements, RadIndex);
%     Ex =
build_imuEx(Nstates,sigmaVx,sigmaVy,sigmaVz,sigmaYdot,Objer
r,Raderr,deltaT);
%     Ez = build_imuEz(Nmeasurements,Nbeams ,[Xerr
YAWerr],ObjMeasErr, RadMeasErr);
     Xk(:,1) = zeros(12,1);
    Xk(1:9,1) = [X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8)
X(9)];
    Xk(10:13,1) = [10 10 2 2000];
%     load('wallandtree.mat')
% xtemp = [5 0 4]';
% xdestemp = [5 9 4]';
% Vdir = (xdestemp - xtemp)/norm(xdestemp - xtemp);
```

```
%     scatter3(xobs-.5,yobs-.5,zobs-.5,obs_str,'b');

%     keyboard;
    P = zeros(Nstates,Nstates);
%     plot3(Xk(1,1:end-1),Xk(2,1:end-1),Xk(3,1:end-1),'b')
    Zk = zeros(Nmeasurements,1);
    Zk(1:4,1) = [Xk(1,1) Xk(2,1) Xk(3,1) Xk(6,1)]';
    tog = 0;
    HeadingState = States(1).X - X;
    Grad = 0;
    %Just initial flight/sensing test routine
    disp('Begin sim');

%     figure;


    Pi = [0 0 0]';
    lpi = 0;
    lq = 0;
    qi = [0 0 0]';
    pf = [0 0 0]';
    qstr = 0;
    ti = 0;
    firstPass = 1;
    landmarkexists = 0;


   NumberOfObservations = detect_env3_imu_cleaned(world, X,
[20 20 20], ppm,t,sensX, sensY, sensZ,Nbeams,Ncubes,Xk);
   [Pi qi pf ti,which_pi_measured lpi lq landmarkexists
qstr Zobj] = Parse_Observations(Pi, lpi, qi, lq, qstr, ti,
t, NumberOfObservations,firstPass,landmarkexists);
   Xk = fix_state_vector(Xk, lpi, Pi);
   P = build_Po(Xk);
   Ex = build_Ex(lpi, XYZERROR,VXYZerr, VRPYerr, ObjErr1,
RadErr1, deltaT);
    firstPass = 0;
    for i = 1:numel(waypoints)     %Navigate to each point

        decision = 0;
        desiredX = States(1).X;     %set desired state
        finalPos = States(1).X(1:3);
        disp('Detecting/Integrating');
```

```
        Vdir = (waypoints(i).X(1:3) -
X(1:3))/norm(waypoints(i).X(1:3) - X(1:3));

        States(1).X = [X(1) X(2) X(3) 0 0 pi/2 Vdes*Vdir(1)
Vdes*Vdir(2) Vdes*Vdir(3) 0 0]';
        currentWay = waypoints(i).X(1:3);

        while (norm(X(1:3)-waypoints(i).X(1:3))>=.5)
%Once within some distance stop move to next point

%             while (t<=30)


%             add_gps_err();
%function to add random walk aspect


            Xkest = TransitionModel(Xk,IMU',deltaT);
            [Ak, ~] = get_Ak_jacobian2(Xk,IMU');
            P = Ak*P*(Ak')+Ex;



            A = RK4(@EOM2,X,t,dt,tsteps,0);
%integrate motion for 20 time steps
            detect_rad(X(1:3),deltaT);

            NumberOfObservations =
detect_env3_imu_cleaned(world, X, [20 20 20], ppm,t,sensX,
sensY, sensZ,Nbeams,Ncubes,Xk);
            [Pi qi pf ti,which_pi_measured lpi lq
landmarkexists qstr Zobj] = Parse_Observations(Pi, lpi, qi,
lq, qstr, ti, t,
NumberOfObservations,firstPass,landmarkexists);
            Xkest = fix_state_vector(Xkest, lpi, Pi);
            P = Fix_Pk(P,Xkest);
            Ex = build_Ex(lpi, XYZERROR,VXYZerr, VRPYerr,
ObjErr1, RadErr1, deltaT);


            if ((mod(t,1) < .00001) || (abs(mod(t,1)-1) <
.0001))      %every 1 second store radiation data
                if (t ~= 0)
                    gotRadMeasurements = 1;
%                     disp('Recording Counts');
```

```
                    record_counts(X(1:3),t);
%record radiation data
%                     disp(X(1:3)');
%                     disp(t);
%                     disp(Xk(length(Xk)-3:length(Xk)-1));
%                     disp (norm(Xk(1:3)-X(1:3)));
%                     disp (RadObs(numRadObs).Counts)
                    rSquared = (Xk(length(Xk)-3) -
Xk(1))^2 + (Xk(length(Xk)-2) - Xk(2))^2 + (Xk(length(Xk)-1)
- Xk(3))^2;
%                     disp(Xk(end)/rSquared);
%                     disp(P(end,end));
                end
%                   keyboard;
%                   RadIndex =
find_measurement_index2_imu(Xkest(:,j), [0 0 0]');
%                   ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,RadObs(end).Counts);
%                     RadIndex = 0;
%                   ZMeas =
handle_Measurements(X,Nmeasurements,[1 1 1 .1],tmpMeas,20);
%             keyboard;
%             else
%                   RadIndex = 0;
%                   ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,0);
            end

            Zest = ObservationModel(pf, which_pi_measured,
Xkest);
            Ez = build_Ez(length(Zest), GPSerr, CompassErr,
ObjErr2, RadErr);
            H =
get_Hk_jacobian(Xkest,pf,which_pi_measured);

            Z = GetMeasurement(X, [GPSerr CompassErr],
Zobj, length(Zest),gotRadMeasurements);

            K = P*transpose(H)*(H*P*transpose(H) + Ez)^(-
1);
            Xk = Xkest + K*(Z-Zest);
            I = eye(length(P));
```

```
            P = (I - K*H)*P;

%             waypoints(i).X(1:2) = Xk(length(Xk)-
3:length(Xk)-2);
%             currentWay = waypoints(i).X(1:3);
            gotRadMeasurements = 0;


            t = t + tsteps*dt;
%store time
            X = A(end,2:12)';                          %store
position
            x1(j) = X(1);
            y1(j) = X(2);
            z1(j) = X(3);
            roll(j) = X(4);
            pitch(j) = X(5);
            yaw(j) = X(6);
            Vx(j) = X(7);
            Vy(j) = X(8);
            Vz(j) = X(9);
            time(j) = t;
            x2(j) = Xk(1);
            y2(j) = Xk(2);
            z2(j) = Xk(3);
            radx(j) = Xk(length(Xk)-3);
            rady(j) = Xk(length(Xk)-2);
            radz(j) = Xk(length(Xk)-1);
            activity(j) = Xk(end);
            gpsX(j) = Z(1);
            gpsY(j) = Z(2);
            gpsZ(j) = Z(3);
            yawEst(j) = Xk(6);
            j = j+1;

            if ((X(1) <=  12) && (X(1) >=  8) && (X(2) >=
9) && (X(2) <=  11)  && (X(3) <=  5))
            wall_collision = 1
            end


%              clf
%               plot3(0,0,0,'r')
%             hold on
```

```
%               axis([-5 5 -5 5 -5 5])


%              hold off
%              refresh
%              pause(.1)
%



               Vdir = (currentWay - X(1:3))/norm(currentWay
- X(1:3));
               heading = atan2(waypoints(i).X(2) -Xk(2),
waypoints(i).X(1) -Xk(1));
                [tog, xcol] = check_traj2(X,currentWay,
Vdir, Pi, lpi);
               if (tog == 1)
%                    disp('Danger');
                   currentWay =
get_new_way2(xcol,X,currentWay,Vdir, Pi, lpi);
%                   currentWay =
get_new_way(xcol,X,currentWay,Vdir, xobs, yobs, zobs);
%                   tog = 0;
               else
                   currentWay = waypoints(i).X(1:3);
               end

              Vdir = (currentWay - X(1:3))/norm(currentWay -
X(1:3));
               if (norm(X(1:3)-waypoints(i).X(1:3))>=.5)
                   Vdes = .5;
               else
                   Vdes = 5;
               end

               States(1).X = [X(1) X(2) X(3) 0 0 heading
Vdes*Vdir(1) Vdes*Vdir(2) Vdes*Vdir(3) 0 0]';
%              States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 0
0 0]';
              %States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 .5
0 0]';
             desiredX = States(1).X;
%              ErrVec = [(x1' -Xk(1,:)') (y1' -Xk(2,:)')
(z1' -Xk(3,:)')];
```

```
%             for p = 1:length(x1)
%                 ErrNorm(p) = norm(ErrVec(p,:));
%             end
%             rms =
sqrt(sum(ErrNorm.*ErrNorm)/length(ErrNorm));
        end

    end

    ErrVec = [(x1' -Xk(1,:)') (y1' -Xk(2,:)') (z1' -
Xk(3,:)')];
    for i = 1:length(x1)
        ErrNorm(i) = norm(ErrVec(i,:));
    end
    rms = sqrt(sum(ErrNorm.*ErrNorm)/length(ErrNorm));
    EstErrX = x1-x2;
    EstErrY = y1-y2;
    EstErrZ = z1-z2;
    GPSErrX = x1 - gpsX;
    GPSErrY = y1 - gpsY;
    GPSErrZ = z1 - gpsZ;
    RadErrX = Sources(1).Pos(1) - radx;
    RadErrY = Sources(1).Pos(2) - rady;
    RadErrZ = Sources(1).Pos(3) - radz;
    DistFromSourceX = Sources(1).Pos(1) - x1;
    DistFromSourceY = Sources(1).Pos(2) - y1;
    DistFromSourceZ = Sources(1).Pos(3) - z1;

    EstErr = sqrt(EstErrX.^2 + EstErrY.^2 + EstErrZ.^2);
    GPSErr = sqrt(GPSErrX.^2 + GPSErrY.^2 + GPSErrZ.^2);
    RadErr = sqrt(RadErrX.^2 + RadErrY.^2 + RadErrZ.^2);
    DistFromSource = sqrt(DistFromSourceX.^2 +
DistFromSourceY.^2 +  DistFromSourceZ.^2);
    for i  =  1:length(EstErr)
        rmsEst(i) = sqrt(sum(EstErr(1:i).^2)/i);
        rmsGPS(i) = sqrt(sum(GPSErr(1:i).^2)/i);
        rmsRad(i) = sqrt(sum(RadErr(1:i).^2)/i);

    end

    [Xveh, Obj, Rad] = break_state_vector(Xk);
```

```
%     figure;     %%Plot trajectory results with real world
imposed
%     plot3(x1,y1,z1,'r');
%     title('Vehicle Trajectory w/ Real World');
%     xlabel('X');
%     ylabel('Y');
%     zlabel('Z');
%     hold on
%     axis([0 20 0 20 0 20]);
%     plot3(xplot,yplot,zplot,'b.')
%     hold off

%     figure
%     plot3(Obj(1,:), Obj(2,:), Obj(3,:),'b.')
%     hold on
%     plot3(x1,y1,z1,'r');
%     title('Vehicle Trajectory w/ Estimated World');
%     xlabel('X');
%     ylabel('Y');
%     zlabel('Z');
%     axis([0 20 0 20 0 20]);
%     hold off


%     figure
%     plot(time,rmsEst,'r--');
%     hold on
%     title('Estimate and GPS Measurement RMS');
%     xlabel('Time (s)');
%     ylabel('RMS (meters)');
%     plot(time,rmsGPS,'b--');
%     legend('Estimated Position RMS', 'GPS Measurement
RMS')
%     hold off

%     figure
%     plot(time, RadErr);
%     title('Radiation Estimation Error Vs. Time (w/
Relative Distance Overlay')
%     xlabel('Time (s)');
%     ylabel('Rad. Err/Distance from Source (meters)');
%     hold on
%     plot(time, DistFromSource,'r--')
```

```
%     legend('Radiation Estimate Difference', 'Distance
from Source');
%     hold off

    char = num2str(RunNumber)
    save(char);
%     keyboard;

end

function motion_RadCovTest(RadGuess,
ERRORS,GPSERROR,RunNumber)
    global Sources
    global RadObs;
    global numRadObs
    global dt;

    global desiredX;
    global SensorStatus
    global decision
    global IMU

    IMU = [0 0 0 0 0 0];
    SensorStatus = [0 0 0 0 0 0];

    global gotRadMeasurements;

    gotRadMeasurements = 0;

    %setup world related variables
    disp('Building World');
    ppm = 5;
    lx = 100;
    ly = lx;
    lz = ly;

    world = build_world_ground(20,20,20,ppm);
    observed_world = build_world_empty(20,20,20,.5);
    world1 = build_world_ground(20,20,20,5);
    [xplot, yplot, zplot] = plot_world(world1, 20, 20, 20,
5,0);
%     figure
%     plot3(xplot,yplot,zplot,'b.')
%     axis([0 20 0 20 0 20])
```

```
%       keyboard
    %set up space for occupancy and radiation cubes
    Occ = cubespace(10,10,10,2);
    Rad = cubespace(10,10,10,2);
    Ncubes =
length(Occ(:,1,1))*length(Occ(1,:,1))*length(Occ(1,1,:));

    States = struct('X',{});
    States(1).X = [10 8 8 0 0 pi/2 0 0 2 0 0]';
    waypoints = struct('X',{});
    waypoints(1).X = [10 12 7]';
       waypoints(2).X = [10 7 7]';
%       waypoints(3).X = [15 5 7]';
%       waypoints(4).X = [12 5 3]';
%       waypoints(5).X = [5 12 5]';
%       waypoints(6).X = [2 3 5]';
%      waypoints(7).X = [9 4 10]';
%      waypoints(8).X = [1 9 5]';
%      waypoints(9).X = [5 9 5]';
%      waypoints(10).X = [1 9 5]';
%      waypoints(11).X = [1 6 5]';
%      waypoints(12).X = [1 15 5]';
%     waypoints(13).X = [2 7 5]';
%     waypoints(14).X = [9 7 5]';
%     waypoints(15).X = [9 8 5]';
%     waypoints(16).X = [2 8 5]';
%     waypoints(17).X = [2 9 5]';
%     waypoints(18).X = [9 9 5]';
%     waypoints(1).X = [2 3 5]';

    Vdes = 2;
    X = [10 6 7 0 0 pi/2 0 0 0 0 0]';
    tsteps = 20;
    t = 0;
    dt = .01;
    deltaT = tsteps*dt;
    j = 1;
    C = 2*pi*5;
    T = C/Vdes;
    f = 1/T;
    w = 2*pi*f;
    States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 5 0 0 0]';

    %States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 .5 0 0]';
```

```
Nstates = length(X) + 2*Ncubes;

%Build some initial errors for Ex
XYZERROR = .02;
VXYZerr = .1;
VRPYerr = .1;
sigmaVx = .1;
sigmaVy = .1;
sigmaVz = .1;
sigmaYdot = .1;
sigmaX = deltaT*sigmaVx;
sigmaY = deltaT*sigmaVx;
sigmaZ = deltaT*sigmaVx;
sigmaRoll = .1;
sigmaPitch = .1;
sigmaYaw = deltaT*sigmaYdot;
ObjErr1 = .5;
RadErr1 = ERRORS;


%errors associated with Ez
GPSerr = GPSERROR;
CompassErr = .1;
ObjErr2 = 1;
RadErr = 200;
Xerr = 0;
Yerr = 1;
Zerr = 1;
YAWerr = 0;
ObjMeasErr = .001;
RadMeasErr = 4000;

tog =1;
lastx = X(1);
lasty = X(2);
lastz = X(3);

deltaT = tsteps*dt;
I = eye(Nstates);


Nstates = 9 + 2*Ncubes;
I = eye(Nstates);
```

```
    [sensX, sensY, sensZ, Nbeams] =
build_sensor_array(5*pi/6, pi/3,5);
    Nmeasurements = 6 + Ncubes + 1;
    ObjIndices = 0;
    RadIndex = 0;

%     Hk = build_imuHk(Nstates,ObjIndices,
Ncubes,Nmeasurements, RadIndex);
%     Ex =
build_imuEx(Nstates,sigmaVx,sigmaVy,sigmaVz,sigmaYdot,Objer
r,Raderr,deltaT);
%     Ez = build_imuEz(Nmeasurements,Nbeams ,[Xerr
YAWerr],ObjMeasErr, RadMeasErr);
     Xk(:,1) = zeros(12,1);
    Xk(1:9,1) = [X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8)
X(9)];
    Xk(10:13,1) = [10 10 2 2000];
    Xk(10:13,1) = RadGuess;
%     load('wallandtree.mat')
% xtemp = [5 0 4]';
% xdestemp = [5 9 4]';
% Vdir = (xdestemp - xtemp)/norm(xdestemp - xtemp);
%     scatter3(xobs-.5,yobs-.5,zobs-.5,obs_str,'b');

%     keyboard;
    P = zeros(Nstates,Nstates);
%     plot3(Xk(1,1:end-1),Xk(2,1:end-1),Xk(3,1:end-1),'b')
    Zk = zeros(Nmeasurements,1);
    Zk(1:4,1) = [Xk(1,1) Xk(2,1) Xk(3,1) Xk(6,1)]';
    tog = 0;
    HeadingState = States(1).X - X;
    Grad = 0;
    %Just initial flight/sensing test routine
    disp('Begin sim');

%     figure;


    Pi = [0 0 0]';
    lpi = 0;
    lq = 0;
    qi = [0 0 0]';
    pf = [0 0 0]';
    qstr = 0;
```

```
    ti = 0;
    firstPass = 1;
    landmarkexists = 0;


   NumberOfObservations = detect_env3_imu_cleaned(world, X,
[20 20 20], ppm,t,sensX, sensY, sensZ,Nbeams,Ncubes,Xk);
   [Pi qi pf ti,which_pi_measured lpi lq landmarkexists
qstr Zobj] = Parse_Observations(Pi, lpi, qi, lq, qstr, ti,
t, NumberOfObservations,firstPass,landmarkexists);
   Xk = fix_state_vector(Xk, lpi, Pi);
   P = build_Po(Xk);
   Ex = build_Ex(lpi, XYZERROR,VXYZerr, VRPYerr, ObjErr1,
RadErr1, deltaT);
    firstPass = 0;
    for i = 1:numel(waypoints)     %Navigate to each point

        decision = 0;
        desiredX = States(1).X;     %set desired state
        finalPos = States(1).X(1:3);
        disp('Detecting/Integrating');

        Vdir = (waypoints(i).X(1:3) -
X(1:3))/norm(waypoints(i).X(1:3) - X(1:3));

        States(1).X = [X(1) X(2) X(3) 0 0 pi/2 Vdes*Vdir(1)
Vdes*Vdir(2) Vdes*Vdir(3) 0 0]';
        currentWay = waypoints(i).X(1:3);

        while (norm(X(1:3)-waypoints(i).X(1:3))>=.5)
%Once within some distance stop move to next point

%             while (t<=30)


%             add_gps_err();
%function to add random walk aspect


            Xkest = TransitionModel(Xk,IMU',deltaT);
            [Ak, ~] = get_Ak_jacobian2(Xk,IMU');
            P = Ak*P*(Ak')+Ex;
```

```
            A = RK4(@EOM2,X,t,dt,tsteps,0);
%integrate motion for 20 time steps
            detect_rad(X(1:3),deltaT);

            NumberOfObservations =
detect_env3_imu_cleaned(world, X, [20 20 20], ppm,t,sensX,
sensY, sensZ,Nbeams,Ncubes,Xk);
            [Pi qi pf ti,which_pi_measured lpi lq
landmarkexists qstr Zobj] = Parse_Observations(Pi, lpi, qi,
lq, qstr, ti, t,
NumberOfObservations,firstPass,landmarkexists);
            Xkest = fix_state_vector(Xkest, lpi, Pi);
            P = Fix_Pk(P,Xkest);
            Ex = build_Ex(lpi, XYZERROR,VXYZerr, VRPYerr,
ObjErr1, RadErr1, deltaT);


            if ((mod(t,1) < .00001) || (abs(mod(t,1)-1) <
.0001))      %every 1 second store radiation data
                if (t ~= 0)
                    gotRadMeasurements = 1;
%                     disp('Recording Counts');
                    record_counts(X(1:3),t);
%record radiation data
%                       disp(X(1:3)');
%                       disp(t);
%                       disp(Xk(length(Xk)-3:length(Xk)-1));
%                       disp (norm(Xk(1:3)-X(1:3)));
%                       disp (RadObs(numRadObs).Counts)
                    rSquared = (Xk(length(Xk)-3) -
Xk(1))^2 + (Xk(length(Xk)-2) - Xk(2))^2 + (Xk(length(Xk)-1)
- Xk(3))^2;
%                       disp(Xk(end)/rSquared);
%                       disp(P(end,end));
                end
%                   keyboard;
%                   RadIndex =
find_measurement_index2_imu(Xkest(:,j), [0 0 0]');
%                   ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,RadObs(end).Counts);
%                       RadIndex = 0;
%                   ZMeas =
handle_Measurements(X,Nmeasurements,[1 1 1 .1],tmpMeas,20);
```

```
%             keyboard;
%             else
%                  RadIndex = 0;
%                  ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,0);
            end

            [Zest ~] = ObservationModel(pf,
which_pi_measured, Xkest);
            Ez = build_Ez(length(Zest), GPSerr, CompassErr,
ObjErr2, RadErr);
            H =
get_Hk_jacobian(Xkest,pf,which_pi_measured);
            if gotRadMeasurements == 1
%                keyboard
            end
            Z = GetMeasurement(X, [GPSerr CompassErr],
Zobj, length(Zest),gotRadMeasurements);

            K = P*transpose(H)*(H*P*transpose(H) + Ez)^(-
1);
            Xk = Xkest + K*(Z-Zest);
            I = eye(length(P));
            P = (I - K*H)*P;

%             waypoints(i).X(1:2) = Xk(length(Xk)-
3:length(Xk)-2);
%             currentWay = waypoints(i).X(1:3);
            gotRadMeasurements = 0;


            t = t + tsteps*dt;
%store time
            X = A(end,2:12)';                          %store
position
            x1(j) = X(1);
            y1(j) = X(2);
            z1(j) = X(3);
            roll(j) = X(4);
            pitch(j) = X(5);
            yaw(j) = X(6);
            Vx(j) = X(7);
            Vy(j) = X(8);
```

```
            Vz(j) = X(9);
            time(j) = t;
            x2(j) = Xk(1);
            y2(j) = Xk(2);
            z2(j) = Xk(3);
            radx(j) = Xk(length(Xk)-3);
            rady(j) = Xk(length(Xk)-2);
            radz(j) = Xk(length(Xk)-1);
            activity(j) = Xk(end);
            gpsX(j) = Z(1);
            gpsY(j) = Z(2);
            gpsZ(j) = Z(3);
            yawEst(j) = Xk(6);
            j = j+1;



%             clf
%              plot3(0,0,0,'r')
%             hold on
%             axis([-5 5 -5 5 -5 5])


%            hold off
%            refresh
%            pause(.1)
%



            Vdir = (currentWay - X(1:3))/norm(currentWay
- X(1:3));
            heading = atan2(waypoints(i).X(2) -Xk(2),
waypoints(i).X(1) -Xk(1));
             [tog, xcol] = check_traj2(X,currentWay,
Vdir, Pi, lpi);
            if (tog == 1)
%                 disp('Danger');
                currentWay =
get_new_way2(xcol,X,currentWay,Vdir, Pi, lpi);
%                 currentWay =
get_new_way(xcol,X,currentWay,Vdir, xobs, yobs, zobs);
%                 tog = 0;
```

```
            else
                currentWay = waypoints(i).X(1:3);
            end

            Vdir = (currentWay - X(1:3))/norm(currentWay -
X(1:3));
            if (norm(X(1:3)-waypoints(i).X(1:3))>=.5)
                Vdes = .5;
            else
                Vdes = 5;
            end

             States(1).X = [X(1) X(2) X(3) 0 0 heading
Vdes*Vdir(1) Vdes*Vdir(2) Vdes*Vdir(3) 0 0]';
%             States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 0
0 0]';
            %States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 .5
0 0]';
          desiredX = States(1).X;
%             ErrVec = [(x1' -Xk(1,:)') (y1' -Xk(2,:)')
(z1' -Xk(3,:)')];
%             for p = 1:length(x1)
%                 ErrNorm(p) = norm(ErrVec(p,:));
%             end
%             rms =
sqrt(sum(ErrNorm.*ErrNorm)/length(ErrNorm));
        end

        if i == 2
            while (t<=28)


%             add_gps_err();
%function to add random walk aspect


            Xkest = TransitionModel(Xk,IMU',deltaT);
            [Ak, ~] = get_Ak_jacobian2(Xk,IMU');
            P = Ak*P*(Ak')+Ex;



            A = RK4(@EOM2,X,t,dt,tsteps,0);
%integrate motion for 20 time steps
```

```
            detect_rad(X(1:3),deltaT);

            NumberOfObservations =
detect_env3_imu_cleaned(world, X, [20 20 20], ppm,t,sensX,
sensY, sensZ,Nbeams,Ncubes,Xk);
            [Pi qi pf ti,which_pi_measured lpi lq
landmarkexists qstr Zobj] = Parse_Observations(Pi, lpi, qi,
lq, qstr, ti, t,
NumberOfObservations,firstPass,landmarkexists);
            Xkest = fix_state_vector(Xkest, lpi, Pi);
            P = Fix_Pk(P,Xkest);
            Ex = build_Ex(lpi, XYZERROR,VXYZerr, VRPYerr,
ObjErr1, RadErr1, deltaT);


            if ((mod(t,1) < .00001) || (abs(mod(t,1)-1) <
.0001))      %every 1 second store radiation data
                if (t ~= 0)
                    gotRadMeasurements = 1;
%                     disp('Recording Counts');
                    record_counts(X(1:3),t);
%record radiation data
%                       disp(X(1:3)');
%                       disp(t);
%                       disp(Xk(length(Xk)-3:length(Xk)-1));
%                       disp (norm(Xk(1:3)-X(1:3)));
%                       disp (RadObs(numRadObs).Counts)
                    rSquared = (Xk(length(Xk)-3) -
Xk(1))^2 + (Xk(length(Xk)-2) - Xk(2))^2 + (Xk(length(Xk)-1)
- Xk(3))^2;
%                       disp(Xk(end)/rSquared);
%                       disp(P(end,end));
                end
%                   keyboard;
%                   RadIndex =
find_measurement_index2_imu(Xkest(:,j), [0 0 0]');
%                   ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,RadObs(end).Counts);
%                       RadIndex = 0;
%                   ZMeas =
handle_Measurements(X,Nmeasurements,[1 1 1 .1],tmpMeas,20);
%             keyboard;
%             else
```

```
%                   RadIndex = 0;
%                   ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,0);
            end

            [Zest ~] = ObservationModel(pf,
which_pi_measured, Xkest);
            Ez = build_Ez(length(Zest), GPSerr, CompassErr,
ObjErr2, RadErr);
            H =
get_Hk_jacobian(Xkest,pf,which_pi_measured);
            if gotRadMeasurements == 1
%                keyboard
            end
            Z = GetMeasurement(X, [GPSerr CompassErr],
Zobj, length(Zest),gotRadMeasurements);

            K = P*transpose(H)*(H*P*transpose(H) + Ez)^(-
1);
            Xk = Xkest + K*(Z-Zest);
            I = eye(length(P));
            P = (I - K*H)*P;

%             waypoints(i).X(1:2) = Xk(length(Xk)-
3:length(Xk)-2);
%             currentWay = waypoints(i).X(1:3);
            gotRadMeasurements = 0;


            t = t + tsteps*dt;
%store time
            X = A(end,2:12)';                          %store
position
            x1(j) = X(1);
            y1(j) = X(2);
            z1(j) = X(3);
            roll(j) = X(4);
            pitch(j) = X(5);
            yaw(j) = X(6);
            Vx(j) = X(7);
            Vy(j) = X(8);
            Vz(j) = X(9);
            time(j) = t;
```

```
            x2(j) = Xk(1);
            y2(j) = Xk(2);
            z2(j) = Xk(3);
            radx(j) = Xk(length(Xk)-3);
            rady(j) = Xk(length(Xk)-2);
            radz(j) = Xk(length(Xk)-1);
            activity(j) = Xk(end);
            gpsX(j) = Z(1);
            gpsY(j) = Z(2);
            gpsZ(j) = Z(3);
            yawEst(j) = Xk(6);
            j = j+1;



%             clf
%              plot3(0,0,0,'r')
%             hold on
%             axis([-5 5 -5 5 -5 5])


%            hold off
%            refresh
%            pause(.1)
%



            Vdir = (currentWay - X(1:3))/norm(currentWay
- X(1:3));
            heading = atan2(waypoints(i).X(2) -Xk(2),
waypoints(i).X(1) -Xk(1));
             [tog, xcol] = check_traj2(X,currentWay,
Vdir, Pi, lpi);
            if (tog == 1)
%                 disp('Danger');
                currentWay =
get_new_way2(xcol,X,currentWay,Vdir, Pi, lpi);
%                 currentWay =
get_new_way(xcol,X,currentWay,Vdir, xobs, yobs, zobs);
%                 tog = 0;
            else
                currentWay = waypoints(i).X(1:3);
```

```
            end

            Vdir = (currentWay - X(1:3))/norm(currentWay -
X(1:3));
            if (norm(X(1:3)-waypoints(i).X(1:3))>=.5)
                Vdes = .5;
            else
                Vdes = 5;
            end

             States(1).X = [X(1) X(2) X(3) 0 0 heading
Vdes*Vdir(1) Vdes*Vdir(2) Vdes*Vdir(3) 0 0]';
%             States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 0
0 0]';
            %States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 .5
0 0]';
           desiredX = States(1).X;
%             ErrVec = [(x1' -Xk(1,:)') (y1' -Xk(2,:)')
(z1' -Xk(3,:)')];
%             for p = 1:length(x1)
%                 ErrNorm(p) = norm(ErrVec(p,:));
%             end
%             rms =
sqrt(sum(ErrNorm.*ErrNorm)/length(ErrNorm));
        end


        end

    end

    ErrVec = [(x1' -Xk(1,:)') (y1' -Xk(2,:)') (z1' -
Xk(3,:)')];
    for i = 1:length(x1)
        ErrNorm(i) = norm(ErrVec(i,:));
    end
    rms = sqrt(sum(ErrNorm.*ErrNorm)/length(ErrNorm));
    EstErrX = x1-x2;
    EstErrY = y1-y2;
    EstErrZ = z1-z2;
    GPSErrX = x1 - gpsX;
    GPSErrY = y1 - gpsY;
    GPSErrZ = z1 - gpsZ;
    RadErrX = Sources(1).Pos(1) - radx;
```

```
    RadErrY = Sources(1).Pos(2) - rady;
    RadErrZ = Sources(1).Pos(3) - radz;
    DistFromSourceX = Sources(1).Pos(1) - x1;
    DistFromSourceY = Sources(1).Pos(2) - y1;
    DistFromSourceZ = Sources(1).Pos(3) - z1;

    EstErr = sqrt(EstErrX.^2 + EstErrY.^2 + EstErrZ.^2);
    GPSErr = sqrt(GPSErrX.^2 + GPSErrY.^2 + GPSErrZ.^2);
    RadErr = sqrt(RadErrX.^2 + RadErrY.^2 + RadErrZ.^2);
    DistFromSource = sqrt(DistFromSourceX.^2 +
DistFromSourceY.^2 +  DistFromSourceZ.^2);
    for i  =  1:length(EstErr)
        rmsEst(i) = sqrt(sum(EstErr(1:i).^2)/i);
        rmsGPS(i) = sqrt(sum(GPSErr(1:i).^2)/i);
        rmsRad(i) = sqrt(sum(RadErr(1:i).^2)/i);

    end

    [Xveh, Obj, Rad] = break_state_vector(Xk);



%     figure;     %%Plot trajectory results with real world
imposed
%     plot3(x1,y1,z1,'r');
%     title('Vehicle Trajectory w/ Real World');
%     xlabel('X');
%     ylabel('Y');
%     zlabel('Z');
%     hold on
%     axis([0 20 0 20 0 20]);
%     plot3(xplot,yplot,zplot,'b.')
%     hold off
%
%      figure
%      plot3(Obj(1,:), Obj(2,:), Obj(3,:),'b.')
%      hold on
%      plot3(x1,y1,z1,'r');
%      title('Vehicle Trajectory w/ Estimated World');
%      xlabel('X');
%      ylabel('Y');
%      zlabel('Z');
%      axis([0 20 0 20 0 20]);
%      hold off
```

```
%
%
%      figure
%      plot(time,EstErr,'r--');
%      hold on
%      title('Estimate and GPS Measurement RMS');
%      xlabel('Time (s)');
%      ylabel('RMS (meters)');
% %      plot(time,rmsGPS,'b--');
% %      legend('Estimated Position RMS', 'GPS Measurement
RMS')
%      hold off
%
%       figure
%       plot(time, RadErr);
%       title('Radiation Estimation Error Vs. Time (w/
Relative Distance Overlay')
%       xlabel('Time (s)');
%       ylabel('Rad. Err/Distance from Source (meters)');
%       hold on
%       plot(time, DistFromSource,'r--')
%       legend('Radiation Estimate Difference', 'Distance
from Source');
%       hold off

      char = num2str(RunNumber)
      save(char);
%        keyboard;

end

function motion_ObsCovTest(WorldType,GPSERROR,RunNumber)
    global Sources
    global RadObs;
    global numRadObs
    global dt;

    global desiredX;
    global SensorStatus
    global decision
    global IMU

    IMU = [0 0 0 0 0 0];
    SensorStatus = [0 0 0 0 0 0];
```

```
    global gotRadMeasurements;

    gotRadMeasurements = 0;

    %setup world related variables
    disp('Building World');
    ppm = 5;
    lx = 100;
    ly = lx;
    lz = ly;

    if WorldType == 0;
        world = build_world_ground(20,20,20,ppm);
        observed_world = build_world_empty(20,20,20,.5);
        world1 = build_world_ground(20,20,20,5);
        [xplot, yplot, zplot] = plot_world(world1, 20, 20,
20, 5,0);
    else
        world = build_world_wall2(20,20,20,ppm);
        observed_world = build_world_empty(20,20,20,.5);
        world1 = build_world_wall2(20,20,20,5);
        [xplot, yplot, zplot] = plot_world(world1, 20, 20,
20, 5,0);
    end
%     figure
%     plot3(xplot,yplot,zplot,'b.')
%     axis([0 20 0 20 0 20])
%      keyboard
    %set up space for occupancy and radiation cubes
    Occ = cubespace(10,10,10,2);
    Rad = cubespace(10,10,10,2);
    Ncubes =
length(Occ(:,1,1))*length(Occ(1,:,1))*length(Occ(1,1,:));

    States = struct('X',{});
    States(1).X = [10 8 8 0 0 pi/2 0 0 2 0 0]';
    waypoints = struct('X',{});
    waypoints(1).X = [5 7 5]';
       waypoints(2).X = [5 2 5]';
%       waypoints(3).X = [15 5 7]';
%       waypoints(4).X = [12 5 3]';
%       waypoints(5).X = [5 12 5]';
%       waypoints(6).X = [2 3 5]';
```

```
%       waypoints(7).X = [9 4 10]';
%       waypoints(8).X = [1 9 5]';
%       waypoints(9).X = [5 9 5]';
%       waypoints(10).X = [1 9 5]';
%       waypoints(11).X = [1 6 5]';
%       waypoints(12).X = [1 15 5]';
%      waypoints(13).X = [2 7 5]';
%      waypoints(14).X = [9 7 5]';
%      waypoints(15).X = [9 8 5]';
%      waypoints(16).X = [2 8 5]';
%      waypoints(17).X = [2 9 5]';
%      waypoints(18).X = [9 9 5]';
%      waypoints(1).X = [2 3 5]';

    Vdes = 2;
    X = [5 2 5 0 0 pi/2 0 0 0 0 0]';
    tsteps = 20;
    t = 0;
    dt = .01;
    deltaT = tsteps*dt;
    j = 1;
    C = 2*pi*5;
    T = C/Vdes;
    f = 1/T;
    w = 2*pi*f;
    States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 5 0 0 0]';

    %States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 .5 0 0]';
    Nstates = length(X) + 2*Ncubes;

    %Build some initial errors for Ex
    XYZERROR = .02;
    VXYZerr = .1;
    VRPYerr = .1;
    sigmaVx = .1;
    sigmaVy = .1;
    sigmaVz = .1;
    sigmaYdot = .1;
    sigmaX = deltaT*sigmaVx;
    sigmaY = deltaT*sigmaVx;
    sigmaZ = deltaT*sigmaVx;
    sigmaRoll = .1;
    sigmaPitch = .1;
    sigmaYaw = deltaT*sigmaYdot;
```

```
    ObjErr1 = .5;
    RadErr1 = [1 1];


    %errors associated with Ez
    GPSerr = GPSERROR;
    CompassErr = .1;
    ObjErr2 = 1;
    RadErr = 200;
    Xerr = 0;
    Yerr = 1;
    Zerr = 1;
    YAWerr = 0;
    ObjMeasErr = .001;
    RadMeasErr = 4000;

    tog =1;
    lastx = X(1);
    lasty = X(2);
    lastz = X(3);

    deltaT = tsteps*dt;
    I = eye(Nstates);


    Nstates = 9 + 2*Ncubes;
    I = eye(Nstates);
    [sensX, sensY, sensZ, Nbeams] =
build_sensor_array(5*pi/6, pi/3,5);
    Nmeasurements = 6 + Ncubes + 1;
    ObjIndices = 0;
    RadIndex = 0;

%     Hk = build_imuHk(Nstates,ObjIndices,
Ncubes,Nmeasurements, RadIndex);
%     Ex =
build_imuEx(Nstates,sigmaVx,sigmaVy,sigmaVz,sigmaYdot,Objer
r,Raderr,deltaT);
%     Ez = build_imuEz(Nmeasurements,Nbeams ,[Xerr
YAWerr],ObjMeasErr, RadMeasErr);
     Xk(:,1) = zeros(12,1);
    Xk(1:9,1) = [X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8)
X(9)];
    Xk(10:13,1) = [10 10 2 0];
```

```
%     Xk(10:13,1) = RadGuess;
%     load('wallandtree.mat')
% xtemp = [5 0 4]';
% xdestemp = [5 9 4]';
% Vdir = (xdestemp - xtemp)/norm(xdestemp - xtemp);
%     scatter3(xobs-.5,yobs-.5,zobs-.5,obs_str,'b');

%     keyboard;
    P = zeros(Nstates,Nstates);
%     plot3(Xk(1,1:end-1),Xk(2,1:end-1),Xk(3,1:end-1),'b')
    Zk = zeros(Nmeasurements,1);
    Zk(1:4,1) = [Xk(1,1) Xk(2,1) Xk(3,1) Xk(6,1)]';
    tog = 0;
    HeadingState = States(1).X - X;
    Grad = 0;
    %Just initial flight/sensing test routine
    disp('Begin sim');

%     figure;


    Pi = [0 0 0]';
    lpi = 0;
    lq = 0;
    qi = [0 0 0]';
    pf = [0 0 0]';
    qstr = 0;
    ti = 0;
    firstPass = 1;
    landmarkexists = 0;


   NumberOfObservations = detect_env3_imu_cleaned(world, X,
[20 20 20], ppm,t,sensX, sensY, sensZ,Nbeams,Ncubes,Xk);
   [Pi qi pf ti,which_pi_measured lpi lq landmarkexists
qstr Zobj] = Parse_Observations(Pi, lpi, qi, lq, qstr, ti,
t, NumberOfObservations,firstPass,landmarkexists);
   Xk = fix_state_vector(Xk, lpi, Pi);
   P = build_Po(Xk);
   Ex = build_Ex(lpi, XYZERROR,VXYZerr, VRPYerr, ObjErr1,
RadErr1, deltaT);
    firstPass = 0;
    for i = 1:numel(waypoints)     %Navigate to each point
```

```
        decision = 0;
        desiredX = States(1).X;     %set desired state
        finalPos = States(1).X(1:3);
        disp('Detecting/Integrating');

        Vdir = (waypoints(i).X(1:3) -
X(1:3))/norm(waypoints(i).X(1:3) - X(1:3));

        States(1).X = [X(1) X(2) X(3) 0 0 pi/2 Vdes*Vdir(1)
Vdes*Vdir(2) Vdes*Vdir(3) 0 0]';
        currentWay = waypoints(i).X(1:3);

        while (norm(X(1:3)-waypoints(i).X(1:3))>=.5)
%Once within some distance stop move to next point

%             while (t<=30)


%             add_gps_err();
%function to add random walk aspect


            Xkest = TransitionModel(Xk,IMU',deltaT);
            [Ak, ~] = get_Ak_jacobian2(Xk,IMU');
            P = Ak*P*(Ak')+Ex;



            A = RK4(@EOM2,X,t,dt,tsteps,0);
%integrate motion for 20 time steps
            detect_rad(X(1:3),deltaT);

            NumberOfObservations =
detect_env3_imu_cleaned(world, X, [20 20 20], ppm,t,sensX,
sensY, sensZ,Nbeams,Ncubes,Xk);
            [Pi qi pf ti,which_pi_measured lpi lq
landmarkexists qstr Zobj] = Parse_Observations(Pi, lpi, qi,
lq, qstr, ti, t,
NumberOfObservations,firstPass,landmarkexists);
            Xkest = fix_state_vector(Xkest, lpi, Pi);
            P = Fix_Pk(P,Xkest);
            Ex = build_Ex(lpi, XYZERROR,VXYZerr, VRPYerr,
ObjErr1, RadErr1, deltaT);
```

```
            if ((mod(t,1) < .00001) || (abs(mod(t,1)-1) <
.0001))       %every 1 second store radiation data
                if (t ~= 0)
                    gotRadMeasurements = 1;
%                     disp('Recording Counts');
                    record_counts(X(1:3),t);
%record radiation data
%                       disp(X(1:3)');
%                       disp(t);
%                       disp(Xk(length(Xk)-3:length(Xk)-1));
%                       disp (norm(Xk(1:3)-X(1:3)));
%                       disp (RadObs(numRadObs).Counts)
                    rSquared = (Xk(length(Xk)-3) -
Xk(1))^2 + (Xk(length(Xk)-2) - Xk(2))^2 + (Xk(length(Xk)-1)
- Xk(3))^2;
%                       disp(Xk(end)/rSquared);
%                       disp(P(end,end));
                end
%                   keyboard;
%                   RadIndex =
find_measurement_index2_imu(Xkest(:,j), [0 0 0]');
%                   ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,RadObs(end).Counts);
%                       RadIndex = 0;
%                   ZMeas =
handle_Measurements(X,Nmeasurements,[1 1 1 .1],tmpMeas,20);
%               keyboard;
%               else
%                    RadIndex = 0;
%                    ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,0);
            end

            [Zest ~] = ObservationModel(pf,
which_pi_measured, Xkest);
            Ez = build_Ez(length(Zest), GPSerr, CompassErr,
ObjErr2, RadErr);
            H =
get_Hk_jacobian(Xkest,pf,which_pi_measured);
            if gotRadMeasurements == 1
%                keyboard
            end
```

```
            Z = GetMeasurement(X, [GPSerr CompassErr],
Zobj, length(Zest),gotRadMeasurements);

            K = P*transpose(H)*(H*P*transpose(H) + Ez)^(-
1);
            Xk = Xkest + K*(Z-Zest);
            I = eye(length(P));
            P = (I - K*H)*P;

%             waypoints(i).X(1:2) = Xk(length(Xk)-
3:length(Xk)-2);
%             currentWay = waypoints(i).X(1:3);
            gotRadMeasurements = 0;


            t = t + tsteps*dt;
%store time
            X = A(end,2:12)';                          %store
position
            x1(j) = X(1);
            y1(j) = X(2);
            z1(j) = X(3);
            roll(j) = X(4);
            pitch(j) = X(5);
            yaw(j) = X(6);
            Vx(j) = X(7);
            Vy(j) = X(8);
            Vz(j) = X(9);
            time(j) = t;
            x2(j) = Xk(1);
            y2(j) = Xk(2);
            z2(j) = Xk(3);
            radx(j) = Xk(length(Xk)-3);
            rady(j) = Xk(length(Xk)-2);
            radz(j) = Xk(length(Xk)-1);
            activity(j) = Xk(end);
            gpsX(j) = Z(1);
            gpsY(j) = Z(2);
            gpsZ(j) = Z(3);
            yawEst(j) = Xk(6);
            j = j+1;
```

```
%               clf
%                plot3(0,0,0,'r')
%               hold on
%               axis([-5 5 -5 5 -5 5])


%              hold off
%              refresh
%              pause(.1)
%



               Vdir = (currentWay - X(1:3))/norm(currentWay
- X(1:3));
               heading = atan2(waypoints(i).X(2) -Xk(2),
waypoints(i).X(1) -Xk(1));
                [tog, xcol] = check_traj2(X,currentWay,
Vdir, Pi, lpi);
               if (tog == 1)
%                    disp('Danger');
                   currentWay =
get_new_way2(xcol,X,currentWay,Vdir, Pi, lpi);
%                   currentWay =
get_new_way(xcol,X,currentWay,Vdir, xobs, yobs, zobs);
%                   tog = 0;
               else
                   currentWay = waypoints(i).X(1:3);
               end

              Vdir = (currentWay - X(1:3))/norm(currentWay -
X(1:3));
              if (norm(X(1:3)-waypoints(i).X(1:3))>=.5)
                  Vdes = .5;
              else
                  Vdes = 5;
              end

               States(1).X = [X(1) X(2) X(3) 0 0 pi/2
Vdes*Vdir(1) Vdes*Vdir(2) Vdes*Vdir(3) 0 0]';
%              States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 0
0 0]';
```

```
            %States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 .5
0 0]';
            desiredX = States(1).X;
%             ErrVec = [(x1' -Xk(1,:)') (y1' -Xk(2,:)')
(z1' -Xk(3,:)')];
%             for p = 1:length(x1)
%                 ErrNorm(p) = norm(ErrVec(p,:));
%             end
%             rms =
sqrt(sum(ErrNorm.*ErrNorm)/length(ErrNorm));
        end

        if i == 2
            while (t<=35)


%             add_gps_err();
%function to add random walk aspect


            Xkest = TransitionModel(Xk,IMU',deltaT);
            [Ak, ~] = get_Ak_jacobian2(Xk,IMU');
            P = Ak*P*(Ak')+Ex;



            A = RK4(@EOM2,X,t,dt,tsteps,0);
%integrate motion for 20 time steps
            detect_rad(X(1:3),deltaT);

            NumberOfObservations =
detect_env3_imu_cleaned(world, X, [20 20 20], ppm,t,sensX,
sensY, sensZ,Nbeams,Ncubes,Xk);
            [Pi qi pf ti,which_pi_measured lpi lq
landmarkexists qstr Zobj] = Parse_Observations(Pi, lpi, qi,
lq, qstr, ti, t,
NumberOfObservations,firstPass,landmarkexists);
            Xkest = fix_state_vector(Xkest, lpi, Pi);
            P = Fix_Pk(P,Xkest);
            Ex = build_Ex(lpi, XYZERROR,VXYZerr, VRPYerr,
ObjErr1, RadErr1, deltaT);
```

```
            if ((mod(t,1) < .00001) || (abs(mod(t,1)-1) <
.0001))       %every 1 second store radiation data
                if (t ~= 0)
                    gotRadMeasurements = 1;
%                     disp('Recording Counts');
                    record_counts(X(1:3),t);
%record radiation data
%                       disp(X(1:3)');
%                       disp(t);
%                       disp(Xk(length(Xk)-3:length(Xk)-1));
%                       disp (norm(Xk(1:3)-X(1:3)));
%                       disp (RadObs(numRadObs).Counts)
                    rSquared = (Xk(length(Xk)-3) -
Xk(1))^2 + (Xk(length(Xk)-2) - Xk(2))^2 + (Xk(length(Xk)-1)
- Xk(3))^2;
%                       disp(Xk(end)/rSquared);
%                       disp(P(end,end));
                end
%                   keyboard;
%                   RadIndex =
find_measurement_index2_imu(Xkest(:,j), [0 0 0]');
%                   ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,RadObs(end).Counts);
%                       RadIndex = 0;
%                   ZMeas =
handle_Measurements(X,Nmeasurements,[1 1 1 .1],tmpMeas,20);
%               keyboard;
%               else
%                    RadIndex = 0;
%                    ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,0);
            end

            [Zest ~] = ObservationModel(pf,
which_pi_measured, Xkest);
            Ez = build_Ez(length(Zest), GPSerr, CompassErr,
ObjErr2, RadErr);
            H =
get_Hk_jacobian(Xkest,pf,which_pi_measured);
            if gotRadMeasurements == 1
%                keyboard
            end
```

```
            Z = GetMeasurement(X, [GPSerr CompassErr],
Zobj, length(Zest),gotRadMeasurements);

            K = P*transpose(H)*(H*P*transpose(H) + Ez)^(-
1);
            Xk = Xkest + K*(Z-Zest);
            I = eye(length(P));
            P = (I - K*H)*P;

%             waypoints(i).X(1:2) = Xk(length(Xk)-
3:length(Xk)-2);
%             currentWay = waypoints(i).X(1:3);
            gotRadMeasurements = 0;


            t = t + tsteps*dt;
%store time
            X = A(end,2:12)';                          %store
position
            x1(j) = X(1);
            y1(j) = X(2);
            z1(j) = X(3);
            roll(j) = X(4);
            pitch(j) = X(5);
            yaw(j) = X(6);
            Vx(j) = X(7);
            Vy(j) = X(8);
            Vz(j) = X(9);
            time(j) = t;
            x2(j) = Xk(1);
            y2(j) = Xk(2);
            z2(j) = Xk(3);
            radx(j) = Xk(length(Xk)-3);
            rady(j) = Xk(length(Xk)-2);
            radz(j) = Xk(length(Xk)-1);
            activity(j) = Xk(end);
            gpsX(j) = Z(1);
            gpsY(j) = Z(2);
            gpsZ(j) = Z(3);
            yawEst(j) = Xk(6);
            j = j+1;
```

```
%               clf
%                plot3(0,0,0,'r')
%               hold on
%               axis([-5 5 -5 5 -5 5])


%              hold off
%              refresh
%              pause(.1)
%



               Vdir = (currentWay - X(1:3))/norm(currentWay
- X(1:3));
               heading = atan2(waypoints(i).X(2) -Xk(2),
waypoints(i).X(1) -Xk(1));
                [tog, xcol] = check_traj2(X,currentWay,
Vdir, Pi, lpi);
               if (tog == 1)
%                    disp('Danger');
                   currentWay =
get_new_way2(xcol,X,currentWay,Vdir, Pi, lpi);
%                   currentWay =
get_new_way(xcol,X,currentWay,Vdir, xobs, yobs, zobs);
%                   tog = 0;
               else
                   currentWay = waypoints(i).X(1:3);
               end

              Vdir = (currentWay - X(1:3))/norm(currentWay -
X(1:3));
              if (norm(X(1:3)-waypoints(i).X(1:3))>=.5)
                  Vdes = .5;
              else
                  Vdes = 0;
              end

               States(1).X = [5 2 5 0 0 pi/2 0 0 0 0 0]';
%              States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 0
0 0]';
              %States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 .5
0 0]';
```

```
            desiredX = States(1).X;
%             ErrVec = [(x1' -Xk(1,:)') (y1' -Xk(2,:)')
(z1' -Xk(3,:)')];
%             for p = 1:length(x1)
%                 ErrNorm(p) = norm(ErrVec(p,:));
%             end
%             rms =
sqrt(sum(ErrNorm.*ErrNorm)/length(ErrNorm));
        end


        end

    end

    ErrVec = [(x1' -Xk(1,:)') (y1' -Xk(2,:)') (z1' -
Xk(3,:)')];
    for i = 1:length(x1)
        ErrNorm(i) = norm(ErrVec(i,:));
    end
    rms = sqrt(sum(ErrNorm.*ErrNorm)/length(ErrNorm));
    EstErrX = x1-x2;
    EstErrY = y1-y2;
    EstErrZ = z1-z2;
    GPSErrX = x1 - gpsX;
    GPSErrY = y1 - gpsY;
    GPSErrZ = z1 - gpsZ;
    RadErrX = Sources(1).Pos(1) - radx;
    RadErrY = Sources(1).Pos(2) - rady;
    RadErrZ = Sources(1).Pos(3) - radz;
    DistFromSourceX = Sources(1).Pos(1) - x1;
    DistFromSourceY = Sources(1).Pos(2) - y1;
    DistFromSourceZ = Sources(1).Pos(3) - z1;

    EstErr = sqrt(EstErrX.^2 + EstErrY.^2 + EstErrZ.^2);
    GPSErr = sqrt(GPSErrX.^2 + GPSErrY.^2 + GPSErrZ.^2);
    RadErr = sqrt(RadErrX.^2 + RadErrY.^2 + RadErrZ.^2);
    DistFromSource = sqrt(DistFromSourceX.^2 +
DistFromSourceY.^2 +  DistFromSourceZ.^2);
    for i  =  1:length(EstErr)
        rmsEst(i) = sqrt(sum(EstErr(1:i).^2)/i);
        rmsGPS(i) = sqrt(sum(GPSErr(1:i).^2)/i);
        rmsRad(i) = sqrt(sum(RadErr(1:i).^2)/i);
```

```
    end

    [Xveh, Obj, Rad] = break_state_vector(Xk);


%     figure;     %%Plot trajectory results with real world
imposed
%      plot3(x1,y1,z1,'r');
%      title('Vehicle Trajectory w/ Real World');
%      xlabel('X');
%      ylabel('Y');
%      zlabel('Z');
%      hold on
%      axis([0 20 0 20 0 20]);
%      plot3(xplot,yplot,zplot,'b.')
%      hold off
% %
%       figure
%       plot3(Obj(1,:), Obj(2,:), Obj(3,:),'b.')
%       hold on
%       plot3(x1,y1,z1,'r');
%       title('Vehicle Trajectory w/ Estimated World');
%       xlabel('X');
%       ylabel('Y');
%       zlabel('Z');
%       axis([0 20 0 20 0 20]);
%       hold off
% %
% %
%      figure
%      plot(time,EstErr,'r--');
%      hold on
%      title('Estimate and GPS Measurement RMS');
%      xlabel('Time (s)');
%      ylabel('RMS (meters)');
% %      plot(time,rmsGPS,'b--');
%      legend('Estimated Position RMS', 'GPS Measurement
RMS')
%      hold off
% %
%       figure
%       plot(time, RadErr);
```

```
%       title('Radiation Estimation Error Vs. Time (w/
Relative Distance Overlay')
%       xlabel('Time (s)');
%       ylabel('Rad. Err/Distance from Source (meters)');
%       hold on
%       plot(time, DistFromSource,'r--')
%       legend('Radiation Estimate Difference', 'Distance
from Source');
%       hold off

     char = num2str(RunNumber)
     save(char);
%         keyboard;

end

function motion_withIMU_EKF_urban()
    global Sources
    global RadObs;
    global numRadObs
    global dt;

    global desiredX;
    global SensorStatus
    global decision
    global IMU

    IMU = [0 0 0 0 0 0];
    SensorStatus = [0 0 0 0 0 0];

    global gotRadMeasurements;

    gotRadMeasurements = 0;

    %setup world related variables
    disp('Building World');
    ppm = 5;
    lx = 100;
    ly = lx;
    lz = ly;

    world = build_world_urban(20,20,20,ppm);
    observed_world = build_world_empty(20,20,20,.5);
    world1 = build_world_urban(20,20,20,2);
```

```
    [xplot, yplot, zplot] = plot_world(world1, 20, 20, 20,
2,0);
    figure
    plot3(xplot,yplot,zplot,'b.')
    axis([0 20 0 20 0 20])
%       keyboard
    %set up space for occupancy and radiation cubes
    Occ = cubespace(10,10,10,2);
    Rad = cubespace(10,10,10,2);
    Ncubes =
length(Occ(:,1,1))*length(Occ(1,:,1))*length(Occ(1,1,:));

    States = struct('X',{});
    States(1).X = [10 8 8 0 0 pi/2 0 0 2 0 0]';
    waypoints = struct('X',{});
    waypoints(1).X = [7 18 5]';
      waypoints(2).X = [7 10 10]';
      waypoints(3).X = [16 10 11]';
       waypoints(4).X = [16 7 5]';
%       waypoints(5).X = [2 5 3]';
%       waypoints(6).X = [2 3 5]';
%      waypoints(7).X = [9 4 10]';
%      waypoints(8).X = [1 9 5]';
%      waypoints(9).X = [5 9 5]';
%      waypoints(10).X = [1 9 5]';
%      waypoints(11).X = [1 6 5]';
%      waypoints(12).X = [1 15 5]';
%     waypoints(13).X = [2 7 5]';
%     waypoints(14).X = [9 7 5]';
%     waypoints(15).X = [9 8 5]';
%     waypoints(16).X = [2 8 5]';
%     waypoints(17).X = [2 9 5]';
%     waypoints(18).X = [9 9 5]';
%     waypoints(1).X = [2 3 5]';

    Vdes = 2;
    X = [7 3 5 0 0 pi/2 0 0 0 0 0]';
    tsteps = 20;
    t = 0;
    dt = .01;
    deltaT = tsteps*dt;
    j = 1;
    C = 2*pi*5;
    T = C/Vdes;
```

```
f = 1/T;
w = 2*pi*f;
States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 5 0 0 0]';

%States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 .5 0 0]';
Nstates = length(X) + 2*Ncubes;

%Build some initial errors for Ex
VXYZerr = .1;
VRPYerr = .1;
sigmaVx = .1;
sigmaVy = .1;
sigmaVz = .1;
sigmaYdot = .1;
sigmaX = deltaT*sigmaVx;
sigmaY = deltaT*sigmaVx;
sigmaZ = deltaT*sigmaVx;
sigmaRoll = .1;
sigmaPitch = .1;
sigmaYaw = deltaT*sigmaYdot;
ObjErr1 = .5;
RadErr1 = [.25 100];


%errors associated with Ez
GPSerr = 1;
CompassErr = .25;
ObjErr2 = 1;
RadErr = 5;
Xerr = 0;
Yerr = 1;
Zerr = 1;
YAWerr = 0;
ObjMeasErr = .001;
RadMeasErr = .5;

tog =1;
lastx = X(1);
lasty = X(2);
lastz = X(3);

deltaT = tsteps*dt;
I = eye(Nstates);
```

```
    Nstates = 9 + 2*Ncubes;
    I = eye(Nstates);
    [sensX, sensY, sensZ, Nbeams] =
build_sensor_array(5*pi/6, pi/3,5);
    Nmeasurements = 6 + Ncubes + 1;
    ObjIndices = 0;
    RadIndex = 0;

%     Hk = build_imuHk(Nstates,ObjIndices,
Ncubes,Nmeasurements, RadIndex);
%     Ex =
build_imuEx(Nstates,sigmaVx,sigmaVy,sigmaVz,sigmaYdot,Objer
r,Raderr,deltaT);
%     Ez = build_imuEz(Nmeasurements,Nbeams ,[Xerr
YAWerr],ObjMeasErr, RadMeasErr);
     Xk(:,1) = zeros(12,1);
    Xk(1:9,1) = [X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8)
X(9)];
    Xk(10:13,1) = [1 1 2 2000];
%     load('wallandtree.mat')
% xtemp = [5 0 4]';
% xdestemp = [5 9 4]';
% Vdir = (xdestemp - xtemp)/norm(xdestemp - xtemp);
%     scatter3(xobs-.5,yobs-.5,zobs-.5,obs_str,'b');

%     keyboard;
    P = zeros(Nstates,Nstates);
%     plot3(Xk(1,1:end-1),Xk(2,1:end-1),Xk(3,1:end-1),'b')
    Zk = zeros(Nmeasurements,1);
    Zk(1:4,1) = [Xk(1,1) Xk(2,1) Xk(3,1) Xk(6,1)]';
    tog = 0;
    HeadingState = States(1).X - X;
    Grad = 0;
    %Just initial flight/sensing test routine
    disp('Begin sim');

    figure;


    Pi = [0 0 0]';
    lpi = 0;
    lq = 0;
    qi = [0 0 0]';
```

```
    pf = [0 0 0]';
    qstr = 0;
    ti = 0;
    firstPass = 1;
    landmarkexists = 0;


   NumberOfObservations = detect_env3_imu_cleaned(world, X,
[20 20 20], ppm,t,sensX, sensY, sensZ,Nbeams,Ncubes,Xk);
   [Pi qi pf ti,which_pi_measured lpi lq landmarkexists
qstr Zobj] = Parse_Observations(Pi, lpi, qi, lq, qstr, ti,
t, NumberOfObservations,firstPass,landmarkexists);
   Xk = fix_state_vector(Xk, lpi, Pi);
   P = build_Po(Xk);
   Ex = build_Ex(lpi, VXYZerr, VRPYerr, ObjErr1, RadErr1,
deltaT);
    firstPass = 0;
    for i = 1:numel(waypoints)     %Navigate to each point

        decision = 0;
        desiredX = States(1).X;     %set desired state
        finalPos = States(1).X(1:3);
        disp('Detecting/Integrating');

        Vdir = (waypoints(i).X(1:3) -
X(1:3))/norm(waypoints(i).X(1:3) - X(1:3));

        States(1).X = [X(1) X(2) X(3) 0 0 pi/2 Vdes*Vdir(1)
Vdes*Vdir(2) Vdes*Vdir(3) 0 0]';
        currentWay = waypoints(i).X(1:3);

        while (norm(X(1:3)-waypoints(i).X(1:3))>=.5)
%Once within some distance stop move to next point

%             while (t<=30)


%             add_gps_err();
%function to add random walk aspect


            Xkest = TransitionModel(Xk,IMU',deltaT);
            [Ak, ~] = get_Ak_jacobian2(Xk,IMU');
            P = Ak*P*(Ak')+Ex;
```

```
            A = RK4(@EOM2,X,t,dt,tsteps,0);
%integrate motion for 20 time steps
            detect_rad(X(1:3),deltaT);

            NumberOfObservations =
detect_env3_imu_cleaned(world, X, [20 20 20], ppm,t,sensX,
sensY, sensZ,Nbeams,Ncubes,Xk);
            [Pi qi pf ti,which_pi_measured lpi lq
landmarkexists qstr Zobj] = Parse_Observations(Pi, lpi, qi,
lq, qstr, ti, t,
NumberOfObservations,firstPass,landmarkexists);
            Xkest = fix_state_vector(Xkest, lpi, Pi);
            P = Fix_Pk(P,Xkest);
            Ex = build_Ex(lpi, VXYZerr, VRPYerr, ObjErr1,
RadErr1, deltaT);


            if ((mod(t,1) < .00001) || (abs(mod(t,1)-1) <
.0001))      %every 1 second store radiation data
                if (t ~= 0)
                    gotRadMeasurements = 1;
                    disp('Recording Counts');
                    record_counts(X(1:3),t);
%record radiation data
                    disp(X(1:3)');
                    disp(t);
                    disp(Xk(length(Xk)-3:length(Xk)-1));
                    disp (norm(Xk(1:3)-X(1:3)));
                    disp (RadObs(numRadObs).Counts)
                    rSquared = (Xk(length(Xk)-3) -
Xk(1))^2 + (Xk(length(Xk)-2) - Xk(2))^2 + (Xk(length(Xk)-1)
- Xk(3))^2;
                    disp(Xk(end)/rSquared);
                    disp(P(end,end));
                end
%                   keyboard;
%                   RadIndex =
find_measurement_index2_imu(Xkest(:,j), [0 0 0]');
%                   ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,RadObs(end).Counts);
```

```
%                       RadIndex = 0;
%                   ZMeas =
handle_Measurements(X,Nmeasurements,[1 1 1 .1],tmpMeas,20);
%               keyboard;
%               else
%                   RadIndex = 0;
%                   ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,0);
            end

            Zest = ObservationModel(pf, which_pi_measured,
Xkest);
            Ez = build_Ez(length(Zest), GPSerr, CompassErr,
ObjErr2, RadErr);
            H =
get_Hk_jacobian(Xkest,pf,which_pi_measured);

            Z = GetMeasurement(X, [GPSerr CompassErr],
Zobj, length(Zest),gotRadMeasurements);

            K = P*transpose(H)*(H*P*transpose(H) + Ez)^(-
1);
            Xk = Xkest + K*(Z-Zest);
            I = eye(length(P));
            P = (I - K*H)*P;

%             waypoints(i).X(1:2) = Xk(length(Xk)-
3:length(Xk)-2);
%             currentWay = waypoints(i).X(1:3);
            gotRadMeasurements = 0;


            t = t + tsteps*dt;
%store time
            X = A(end,2:12)';                       %store
position
            x1(j) = X(1);
            y1(j) = X(2);
            z1(j) = X(3);
            roll(j) = X(4);
            pitch(j) = X(5);
            yaw(j) = X(6);
            Vx(j) = X(7);
```

```
            Vy(j) = X(8);
            Vz(j) = X(9);
            time(j) = t;
            x2(j) = Xk(1);
            y2(j) = Xk(2);
            z2(j) = Xk(3);
            radx(j) = Xk(length(Xk)-3);
            rady(j) = Xk(length(Xk)-2);
            radz(j) = Xk(length(Xk)-1);
            activity(j) = Xk(end);
            gpsX(j) = Z(1);
            gpsY(j) = Z(2);
            gpsZ(j) = Z(3);
            compass(j) = Z(6);
            yawEst(j) = Xk(6);
            j = j+1;



%               clf
%                plot3(0,0,0,'r')
%               hold on
%               axis([-5 5 -5 5 -5 5])


%              hold off
%              refresh
%              pause(.1)
%



              Vdir = (currentWay - X(1:3))/norm(currentWay
- X(1:3));
              heading = atan2(waypoints(i).X(2) -Xk(2),
waypoints(i).X(1) -Xk(1));
%               [tog, xcol] = check_traj2(X,currentWay,
Vdir, Pi, lpi);
              if (tog == 1)
%                 disp('Danger');
                  currentWay =
get_new_way2(xcol,X,currentWay,Vdir, Pi, lpi);
```

```
%                   currentWay =
get_new_way(xcol,X,currentWay,Vdir, xobs, yobs, zobs);
%                   tog = 0;
                else
                    currentWay = waypoints(i).X(1:3);
                end

               Vdir = (currentWay - X(1:3))/norm(currentWay -
X(1:3));
               if (norm(X(1:3)-waypoints(i).X(1:3))>=.5)
                   Vdes = .5;
               else
                   Vdes = 5;
               end

                States(1).X = [X(1) X(2) X(3) 0 0 heading
Vdes*Vdir(1) Vdes*Vdir(2) Vdes*Vdir(3) 0 0]';
%               States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 0
0 0]';
               %States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 .5
0 0]';
             desiredX = States(1).X;
%              ErrVec = [(x1' -Xk(1,:)') (y1' -Xk(2,:)')
(z1' -Xk(3,:)')];
%               for p = 1:length(x1)
%                   ErrNorm(p) = norm(ErrVec(p,:));
%               end
%               rms =
sqrt(sum(ErrNorm.*ErrNorm)/length(ErrNorm));
        end

    end

    ErrVec = [(x1' -Xk(1,:)') (y1' -Xk(2,:)') (z1' -
Xk(3,:)')];
    for i = 1:length(x1)
        ErrNorm(i) = norm(ErrVec(i,:));
    end
    rms = sqrt(sum(ErrNorm.*ErrNorm)/length(ErrNorm));
    EstErrX = x1-x2;
    EstErrY = y1-y2;
    EstErrZ = z1-z2;
    GPSErrX = x1 - gpsX;
    GPSErrY = y1 - gpsY;
```

```
    GPSErrZ = z1 - gpsZ;
    RadErrX = Sources(1).Pos(1) - radx;
    RadErrY = Sources(1).Pos(2) - rady;
    RadErrZ = Sources(1).Pos(3) - radz;
    DistFromSourceX = Sources(1).Pos(1) - x1;
    DistFromSourceY = Sources(1).Pos(2) - y1;
    DistFromSourceZ = Sources(1).Pos(3) - z1;

    EstErr = sqrt(EstErrX.^2 + EstErrY.^2 + EstErrZ.^2);
    GPSErr = sqrt(GPSErrX.^2 + GPSErrY.^2 + GPSErrZ.^2);
    RadErr = sqrt(RadErrX.^2 + RadErrY.^2 + RadErrZ.^2);
    DistFromSource = sqrt(DistFromSourceX.^2 +
DistFromSourceY.^2 +  DistFromSourceZ.^2);
    for i  =  1:length(EstErr)
        rmsEst(i) = sqrt(sum(EstErr(1:i).^2)/i);
        rmsGPS(i) = sqrt(sum(GPSErr(1:i).^2)/i);
        rmsRad(i) = sqrt(sum(RadErr(1:i).^2)/i);

    end

    [Xveh, Obj, Rad] = break_state_vector(Xk);



    figure;     %%Plot trajectory results with real world
imposed
    plot3(x1,y1,z1,'r');
    title('Vehicle Trajectory w/ Real World');
    xlabel('X');
    ylabel('Y');
    zlabel('Z');
    hold on
    axis([0 20 0 20 4 20]);
    plot3(xplot,yplot,zplot,'b.')
    hold off

    figure
    plot3(Obj(1,:), Obj(2,:), Obj(3,:),'b.')
    hold on
    plot3(x1,y1,z1,'r');
    plot3(x2,y2,z2,'g');
    legend('Landmark Observations','Actual Vehicle
Trajectory','Estimated Vehicle Trajectory');
    title('Vehicle Trajectories w/ Estimated World');
```

```
    xlabel('X (m)');
    ylabel('Y (m)');
    zlabel('Z (m)');
    axis([0 20 0 20 0 20]);
    hold off


    figure
    plot(time,rmsEst,'r--');
    hold on
    title('Estimate and GPS Measurement RMS');
    xlabel('Time (s)');
    ylabel('RMS (meters)');
    plot(time,rmsGPS,'b--');
    legend('Estimated Position RMS', 'GPS Measurement RMS')
    hold off

    figure
    plot(time, RadErr);
    title('Radiation Estimation Error Vs. Time (w/ Relative
Distance Overlay')
    xlabel('Time (s)');
    ylabel('Rad. Err/Distance from Source (meters)');
    hold on
    plot(time, DistFromSource,'r--')
    legend('Radiation Estimate Difference', 'Distance from
Source');
    hold off


    keyboard;

end

function
motion_LoiterSourceDetection(RADERR,RADERR2,Position,RunNum
ber)
    global Sources
    global RadObs;
    global numRadObs
    global dt;

    global desiredX;
    global SensorStatus
```

```
    global decision
    global IMU

    IMU = [0 0 0 0 0 0];
    SensorStatus = [0 0 0 0 0 0];

    global gotRadMeasurements;

    gotRadMeasurements = 0;

    %setup world related variables
    disp('Building World');
    ppm = 5;
    lx = 100;
    ly = lx;
    lz = ly;

    world = build_world_ground(20,20,20,ppm);
    observed_world = build_world_empty(20,20,20,.5);
    world1 = build_world_ground(20,20,20,5);
    [xplot, yplot, zplot] = plot_world(world1, 20, 20, 20,
5,0);
%     figure
%     plot3(xplot,yplot,zplot,'b.')
%     axis([0 20 0 20 0 20])
%      keyboard
    %set up space for occupancy and radiation cubes
    Occ = cubespace(10,10,10,2);
    Rad = cubespace(10,10,10,2);
    Ncubes =
length(Occ(:,1,1))*length(Occ(1,:,1))*length(Occ(1,1,:));

    States = struct('X',{});
    States(1).X = [Position(1) Position(2) Position(3) 0 0
0 0 0 0 0 0]';
    waypoints = struct('X',{});
    waypoints(1).X = [10 15 3]';
%       waypoints(2).X = [15 3 5]';
%       waypoints(3).X = [15 5 7]';
%       waypoints(4).X = [12 5 3]';
%       waypoints(5).X = [5 12 5]';
%       waypoints(6).X = [2 3 5]';
%      waypoints(7).X = [9 4 10]';
%      waypoints(8).X = [1 9 5]';
```

```
%      waypoints(9).X = [5 9 5]';
%      waypoints(10).X = [1 9 5]';
%      waypoints(11).X = [1 6 5]';
%      waypoints(12).X = [1 15 5]';
%     waypoints(13).X = [2 7 5]';
%     waypoints(14).X = [9 7 5]';
%     waypoints(15).X = [9 8 5]';
%     waypoints(16).X = [2 8 5]';
%     waypoints(17).X = [2 9 5]';
%     waypoints(18).X = [9 9 5]';
%     waypoints(1).X = [2 3 5]';

    Vdes = 2;
    X = [Position(1) Position(2) Position(3) 0 0 pi/2 0 0 0
0 0]';
    tsteps = 20;
    t = 0;
    dt = .01;
    deltaT = tsteps*dt;
    j = 1;
    C = 2*pi*5;
    T = C/Vdes;
    f = 1/T;
    w = 2*pi*f;
    States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 5 0 0 0]';

    %States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 .5 0 0]';
    Nstates = length(X) + 2*Ncubes;

    %Build some initial errors for Ex
    XYZERROR = .02;
    VXYZerr = .1;
    VRPYerr = .1;
    sigmaVx = .1;
    sigmaVy = .1;
    sigmaVz = .1;
    sigmaYdot = .1;
    sigmaX = deltaT*sigmaVx;
    sigmaY = deltaT*sigmaVx;
    sigmaZ = deltaT*sigmaVx;
    sigmaRoll = .1;
    sigmaPitch = .1;
    sigmaYaw = deltaT*sigmaYdot;
    ObjErr1 = .5;
```

```
    RadErr1 = RADERR;


    %errors associated with Ez
    GPSerr = 1;
    CompassErr = .1;
    ObjErr2 = 1;
    RadErr = RADERR2;
    Xerr = 0;
    Yerr = 1;
    Zerr = 1;
    YAWerr = 0;
    ObjMeasErr = .001;
    RadMeasErr = .5;

    tog =1;
    lastx = X(1);
    lasty = X(2);
    lastz = X(3);

    deltaT = tsteps*dt;
    I = eye(Nstates);


    Nstates = 9 + 2*Ncubes;
    I = eye(Nstates);
    [sensX, sensY, sensZ, Nbeams] =
build_sensor_array(5*pi/6, pi/3,5);
    Nmeasurements = 6 + Ncubes + 1;
    ObjIndices = 0;
    RadIndex = 0;

%     Hk = build_imuHk(Nstates,ObjIndices,
Ncubes,Nmeasurements, RadIndex);
%     Ex =
build_imuEx(Nstates,sigmaVx,sigmaVy,sigmaVz,sigmaYdot,Objer
r,Raderr,deltaT);
%     Ez = build_imuEz(Nmeasurements,Nbeams ,[Xerr
YAWerr],ObjMeasErr, RadMeasErr);
     Xk(:,1) = zeros(12,1);
    Xk(1:9,1) = [X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8)
X(9)];
    Xk(10:13,1) = [5 5 2 4000];
%     load('wallandtree.mat')
```

```
% xtemp = [5 0 4]';
% xdestemp = [5 9 4]';
% Vdir = (xdestemp - xtemp)/norm(xdestemp - xtemp);
%     scatter3(xobs-.5,yobs-.5,zobs-.5,obs_str,'b');

%     keyboard;
    P = zeros(Nstates,Nstates);
%     plot3(Xk(1,1:end-1),Xk(2,1:end-1),Xk(3,1:end-1),'b')
    Zk = zeros(Nmeasurements,1);
    Zk(1:4,1) = [Xk(1,1) Xk(2,1) Xk(3,1) Xk(6,1)]';
    tog = 0;
    HeadingState = States(1).X - X;
    Grad = 0;
    %Just initial flight/sensing test routine
    disp('Begin sim');

    figure;


    Pi = [0 0 0]';
    lpi = 0;
    lq = 0;
    qi = [0 0 0]';
    pf = [0 0 0]';
    qstr = 0;
    ti = 0;
    firstPass = 1;
    landmarkexists = 0;


   NumberOfObservations = detect_env3_imu_cleaned(world, X,
[20 20 20], ppm,t,sensX, sensY, sensZ,Nbeams,Ncubes,Xk);
   [Pi qi pf ti,which_pi_measured lpi lq landmarkexists
qstr Zobj] = Parse_Observations(Pi, lpi, qi, lq, qstr, ti,
t, NumberOfObservations,firstPass,landmarkexists);
   Xk = fix_state_vector(Xk, lpi, Pi);
   P = build_Po(Xk);
   Ex = build_Ex(lpi, XYZERROR,VXYZerr, VRPYerr, ObjErr1,
RadErr1, deltaT);
    firstPass = 0;
    for i = 1:numel(waypoints)     %Navigate to each point

        decision = 0;
        desiredX = States(1).X;     %set desired state
```

```
        finalPos = States(1).X(1:3);
        disp('Detecting/Integrating');

        Vdir = (waypoints(i).X(1:3) -
X(1:3))/norm(waypoints(i).X(1:3) - X(1:3));

        States(1).X = [X(1) X(2) X(3) 0 0 0 0 0 0 0 0]';
        currentWay = waypoints(i).X(1:3);

%         while (norm(X(1:3)-waypoints(i).X(1:3))>=.5)
%Once within some distance stop move to next point

        while (t<=40)


%             add_gps_err();
%function to add random walk aspect


            Xkest = TransitionModel(Xk,IMU',deltaT);
            [Ak, ~] = get_Ak_jacobian2(Xk,IMU');
            P = Ak*P*(Ak')+Ex;



            A = RK4(@EOM2,X,t,dt,tsteps,0);
%integrate motion for 20 time steps
            detect_rad(X(1:3),deltaT);

            NumberOfObservations =
detect_env3_imu_cleaned(world, X, [20 20 20], ppm,t,sensX,
sensY, sensZ,Nbeams,Ncubes,Xk);
            [Pi qi pf ti,which_pi_measured lpi lq
landmarkexists qstr Zobj] = Parse_Observations(Pi, lpi, qi,
lq, qstr, ti, t,
NumberOfObservations,firstPass,landmarkexists);
            Xkest = fix_state_vector(Xkest, lpi, Pi);
            P = Fix_Pk(P,Xkest);
            Ex = build_Ex(lpi, XYZERROR,VXYZerr, VRPYerr,
ObjErr1, RadErr1, deltaT);


            if ((mod(t,1) < .00001) || (abs(mod(t,1)-1) <
.0001))      %every 1 second store radiation data
                if (t ~= 0)
```

```
                    gotRadMeasurements = 1;
                    disp('Recording Counts');
                    record_counts(X(1:3),t);
%record radiation data
                    disp(X(1:3)');
                    disp(t);
                    disp(Xk(length(Xk)-3:length(Xk)-1));
                    disp (norm(Xk(1:3)-X(1:3)));
                    disp (RadObs(numRadObs).Counts)
                    rSquared = (Xk(length(Xk)-3) -
Xk(1))^2 + (Xk(length(Xk)-2) - Xk(2))^2 + (Xk(length(Xk)-1)
- Xk(3))^2;
                    disp(Xk(end)/rSquared);
                    disp(P(end,end));
                end
%                   keyboard;
%                   RadIndex =
find_measurement_index2_imu(Xkest(:,j), [0 0 0]');
%                   ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,RadObs(end).Counts);
%                       RadIndex = 0;
%                   ZMeas =
handle_Measurements(X,Nmeasurements,[1 1 1 .1],tmpMeas,20);
%               keyboard;
%               else
%                   RadIndex = 0;
%                   ZMeas =
handle_Measurements_imu(X,Nmeasurements,[.0 .0 .0 .0 .0
.0],tmpMeas,0);
            end

            Zest = ObservationModel(pf, which_pi_measured,
Xkest);
            Ez = build_Ez(length(Zest), GPSerr, CompassErr,
ObjErr2, RadErr);
            H =
get_Hk_jacobian(Xkest,pf,which_pi_measured);

            Z = GetMeasurement(X, [GPSerr CompassErr],
Zobj, length(Zest),gotRadMeasurements);

            K = P*transpose(H)*(H*P*transpose(H) + Ez)^(-
1);
```

```
            Xk = Xkest + K*(Z-Zest);
            I = eye(length(P));
            P = (I - K*H)*P;

%             waypoints(i).X(1:2) = Xk(length(Xk)-
3:length(Xk)-2);
%             currentWay = waypoints(i).X(1:3);
            gotRadMeasurements = 0;


            t = t + tsteps*dt;
%store time
            X = A(end,2:12)';                          %store
position
            x1(j) = X(1);
            y1(j) = X(2);
            z1(j) = X(3);
            roll(j) = X(4);
            pitch(j) = X(5);
            yaw(j) = X(6);
            Vx(j) = X(7);
            Vy(j) = X(8);
            Vz(j) = X(9);
            time(j) = t;
            x2(j) = Xk(1);
            y2(j) = Xk(2);
            z2(j) = Xk(3);
            radx(j) = Xk(length(Xk)-3);
            rady(j) = Xk(length(Xk)-2);
            radz(j) = Xk(length(Xk)-1);
            activity(j) = Xk(end);
            gpsX(j) = Z(1);
            gpsY(j) = Z(2);
            gpsZ(j) = Z(3);
            yawEst(j) = Xk(6);
            j = j+1;



%             clf
%              plot3(0,0,0,'r')
%             hold on
%             axis([-5 5 -5 5 -5 5])
```

```
%              hold off
%              refresh
%              pause(.1)
%



               Vdir = (currentWay - X(1:3))/norm(currentWay
- X(1:3));
               heading = atan2(waypoints(i).X(2) -Xk(2),
waypoints(i).X(1) -Xk(1));
                [tog, xcol] = check_traj2(X,currentWay,
Vdir, Pi, lpi);
               if (tog == 1)
                   disp('Danger');
                    currentWay =
get_new_way2(xcol,X,currentWay,Vdir, Pi, lpi);
%                   currentWay =
get_new_way(xcol,X,currentWay,Vdir, xobs, yobs, zobs);
%                   tog = 0;
               else
                   currentWay = waypoints(i).X(1:3);
               end

              Vdir = (currentWay - X(1:3))/norm(currentWay -
X(1:3));
               if (norm(X(1:3)-waypoints(i).X(1:3))>=.5)
                   Vdes = .5;
               else
                   Vdes = 5;
               end

               States(1).X = [X(1) X(2) X(3) 0 0 0 0 0 0 0
0]';
%              States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 0
0 0]';
              %States(1).X = [X(1) X(2) X(3) 0 0 pi/2 0 0 .5
0 0]';
             desiredX = States(1).X;
%              ErrVec = [(x1' -Xk(1,:)') (y1' -Xk(2,:)')
(z1' -Xk(3,:)')];
%              for p = 1:length(x1)
```

```
%                 ErrNorm(p) = norm(ErrVec(p,:));
%             end
%             rms =
sqrt(sum(ErrNorm.*ErrNorm)/length(ErrNorm));
        end

    end

    ErrVec = [(x1' -Xk(1,:)') (y1' -Xk(2,:)') (z1' -
Xk(3,:)')];
    for i = 1:length(x1)
        ErrNorm(i) = norm(ErrVec(i,:));
    end
    rms = sqrt(sum(ErrNorm.*ErrNorm)/length(ErrNorm));
    EstErrX = x1-x2;
    EstErrY = y1-y2;
    EstErrZ = z1-z2;
    GPSErrX = x1 - gpsX;
    GPSErrY = y1 - gpsY;
    GPSErrZ = z1 - gpsZ;
    RadErrX = Sources(1).Pos(1) - radx;
    RadErrY = Sources(1).Pos(2) - rady;
    RadErrZ = Sources(1).Pos(3) - radz;
    DistFromSourceX = Sources(1).Pos(1) - x1;
    DistFromSourceY = Sources(1).Pos(2) - y1;
    DistFromSourceZ = Sources(1).Pos(3) - z1;

    EstErr = sqrt(EstErrX.^2 + EstErrY.^2 + EstErrZ.^2);
    GPSErr = sqrt(GPSErrX.^2 + GPSErrY.^2 + GPSErrZ.^2);
    RadErr = sqrt(RadErrX.^2 + RadErrY.^2 + RadErrZ.^2);
    DistFromSource = sqrt(DistFromSourceX.^2 +
DistFromSourceY.^2 +  DistFromSourceZ.^2);
    for i  =  1:length(EstErr)
        rmsEst(i) = sqrt(sum(EstErr(1:i).^2)/i);
        rmsGPS(i) = sqrt(sum(GPSErr(1:i).^2)/i);
        rmsRad(i) = sqrt(sum(RadErr(1:i).^2)/i);

    end

    [Xveh, Obj, Rad] = break_state_vector(Xk);
```

```
%     figure;     %%Plot trajectory results with real world
imposed
%     plot3(x1,y1,z1,'r');
%     title('Vehicle Trajectory w/ Real World');
%     xlabel('X');
%     ylabel('Y');
%     zlabel('Z');
%     hold on
%     axis([0 20 0 20 0 20]);
%     plot3(xplot,yplot,zplot,'b.')
%     hold off
%
%      figure
%      plot3(Obj(1,:), Obj(2,:), Obj(3,:),'b.')
%      hold on
%      plot3(x1,y1,z1,'r');
%      title('Vehicle Trajectory w/ Estimated World');
%      xlabel('X');
%      ylabel('Y');
%      zlabel('Z');
%      axis([0 20 0 20 0 20]);
%      hold off
%
%
%     figure
%     plot(time,rmsEst,'r--');
%     hold on
%     title('Estimate and GPS Measurement RMS');
%     xlabel('Time (s)');
%     ylabel('RMS (meters)');
%     plot(time,rmsGPS,'b--');
%     legend('Estimated Position RMS', 'GPS Measurement
RMS')
%     hold off
%
%      figure
%      plot(time, RadErr);
%      title('Radiation Estimation Error Vs. Time (w/
Relative Distance Overlay')
%      xlabel('Time (s)');
%      ylabel('Rad. Err/Distance from Source (meters)');
%      hold on
%      plot(time, DistFromSource,'r--')
```

```
%      legend('Radiation Estimate Difference', 'Distance
from Source');
%      hold off

    char = num2str(RunNumber)
    save(char);
%       keyboard;

end

function rates = EOM(t,f)
  %%This function is the rates function that defines the
motion of the
  %%drone without controlled state vectors, it is used by
the controls
  %%function to calculate the effect each state and control
state affect
  %%the aircrafts motion.

    U = f(12:16);

    r1 = [.2 .2 0]';
    r2 = [-.2 .2 0]';
    r3 = [.2 -.2 0]';
    r4 = [-.2 -.2 0]';
    I = [.36 0 0; 0 .36 0; 0 0 .36];

    m = 4.5;%kg
    g = 9.8;%m/s/s
    weight =  m*g;

    DCM3 = [cos(f(6)) sin(f(6)) 0; -sin(f(6)) cos(f(6)) 0;
0 0 1];
    DCM2 = [cos(f(5)) 0 -sin(f(5)); 0 1 0; sin(f(5)) 0
cos(f(5))];
    DCM1 = [1 0 0; 0 cos(f(4)) sin(f(4)); 0 -sin(f(4))
cos(f(4))];

    eRb = DCM1*DCM2*DCM3;

    eFg = [0 0 -weight]';
    bFg = eRb*eFg;

    T1 = [0 0 U(1)]';
```

```
    T2 = [0 0 U(2)]';
    T3 = [0 0 U(3)]';
    T4 = [0 0 U(4)]';

    NetForb = bFg + T1 + T2 + T3 + T4;
    ab = NetForb/m;
    a = transpose(eRb)*ab;

    tor1 = cross(r1,T1);
    tor2 = cross(r2,T2);
    tor3 = cross(r3,T3);
    tor4 = cross(r4,T4);

    NetTor = tor1 + tor2 + tor3 + tor4;
    alpha = NetTor'/I;

    rates = [f(7) f(8) f(9) f(10) f(11) U(5) a(1) a(2) a(3)
alpha(1) alpha(2)]';

end

function rates = EOM2(t,f)
%%This function is the core of the vehicle motion, it used
in the
%%integration process to determine the vehicle's motion.
It computes the
%%rates of the state vector using drone mechanics and
incorporates an LQR
%%controller to navigate to specified coordinates with a
desired attitude.

    global Uo;
    global desiredX;
    global IMU
    r1 = [.2 .2 0]';
    r2 = [-.2 .2 0]';
    r3 = [.2 -.2 0]';
    r4 = [-.2 -.2 0]';
    I = [.36 0 0; 0 .36 0; 0 0 .36];

    m = 4.5;%kg
    g = 9.8;%m/s/s
    weight =  m*g;
```

```
%     detect_rad(f(1:3)');  %This integrates the counts

    k = controls([f Uo']');
    deltaX = f' - desiredX;
    %deltaX(7:9) = deltaX(7:9)*2;
    deltaU = -k*deltaX;
    U = Uo + deltaU;


    DCM3 = [cos(f(6)) sin(f(6)) 0; -sin(f(6)) cos(f(6)) 0;
0 0 1];
    DCM2 = [cos(f(5)) 0 -sin(f(5)); 0 1 0; sin(f(5)) 0
cos(f(5))];
    DCM1 = [1 0 0; 0 cos(f(4)) sin(f(4)); 0 -sin(f(4))
cos(f(4))];

    eRb = DCM1*DCM2*DCM3;

    eFg = [0 0 -weight]';
    bFg = eRb*eFg;

    T1 = [0 0 U(1)]';
    T2 = [0 0 U(2)]';
    T3 = [0 0 U(3)]';
    T4 = [0 0 U(4)]';

    NetForb = bFg + T1 + T2 + T3 + T4;
    ab = NetForb/m;
    a = transpose(eRb)*ab;
    hold on
    tor1 = cross(r1,T1);
    tor2 = cross(r2,T2);
    tor3 = cross(r3,T3);
    tor4 = cross(r4,T4);

    NetTor = tor1 + tor2 + tor3 + tor4;
    alpha = NetTor'/I;
    IMU = [a(1)+normrnd(0,.1) a(2)+normrnd(0,.1)
a(3)+normrnd(0,.1) f(10)+normrnd(0,.1) f(11)+normrnd(0,.1)
U(5)+normrnd(0,.1)];
    rates = [f(7) f(8) f(9) f(10) f(11) U(5) a(1) a(2) a(3)
alpha(1) alpha(2)]';

end
```

```
function k = controls(xi)
%%This function computes the gain matrix Kcontrol for the
LQR controller using
%%numjac and lqr function.
    weight  = 9.8*4.5;
    ti = 1;
    %Get state rates
    xdoti = EOM(ti,xi);
    thresh = 1e-6;
    %Call numjac to compute dFdX
    [dFdX,fac] = numjac(@EOM,ti,xi,xdoti,thresh,[],0);
    %Extract A and B matrices from dFdX
    A = [dFdX(:,1) dFdX(:,2) dFdX(:,3) dFdX(:,4) dFdX(:,5)
dFdX(:,6) dFdX(:,7) dFdX(:,8) dFdX(:,9) dFdX(:,10)
dFdX(:,11) ];
    B = [dFdX(:,12) dFdX(:,13) dFdX(:,14) dFdX(:,15)
dFdX(:,16)];

    %estimated values of the max state values.
    x1max = 1.1*25;
    x2max = 1.1*25;
    x3max = 1.1*25;
    x4max = 1.1*pi/6;
    x5max = 1.1*pi/6;
    x6max = 1.1*pi/6;
    x7max = 8;
    x8max = 8;
    x9max = 5;
    x10max = pi/4;
    x11max = pi/4;
    u5max = pi/4;
    u1max = 1.5*weight;
    u2max = 1.5*weight;
    u3max = 1.5*weight;
    u4max = 1.5*weight;

    %Compute Q and R matrices for lqr
    Q = [1/(x1max^2) 0 0 0 0 0 0 0 0 0 0 ;
        0 1/(x2max^2) 0 0 0 0 0 0 0 0 0 ;
        0 0 1/(x3max^2) 0 0 0 0 0 0 0 0 ;
        0 0 0 1/(x4max^2) 0 0 0 0 0 0 0 ;
        0 0 0 0 1/(x5max^2) 0 0 0 0 0 0 ;
        0 0 0 0 0 1/(x6max^2) 0 0 0 0 0 ;
```

```
        0 0 0 0 0 0 1/(x7max^2) 0 0 0 0 ;
        0 0 0 0 0 0 0 1/(x8max^2) 0 0 0 ;
        0 0 0 0 0 0 0 0 1/(x9max^2) 0 0 ;
        0 0 0 0 0 0 0 0 0 1/(x10max^2) 0 ;
        0 0 0 0 0 0 0 0 0 0 1/(x11max^2) ;
       ];
    R = [1/(u1max^2) 0 0 0 0;
        0 1/(u2max^2) 0 0 0;
        0 0 1/(u3max^2) 0 0;
        0 0 0 1/(u4max^2) 0
        0 0 0 0 1/(u5max^2)];

    %Call lqr function to compute the gain matrix Kcontrol
    [k,S,e] = lqr(A,B,Q,R);



end

function world = build_world_wall(x,y,z,ppm)
%%This function constructs a world with specified X,Y,Z
dimensions.  It
%%creates a custom environment.  ppm variable represents
the resolution of
%%the world occupancy grid, for example a ppm value of 5
will create a
%%world occupany array with 5 elements for every meter of
distance.  A
%%higher ppm allows for a finer resolution of object
location but comes
%%with a price of increased memory and computation.

    world = zeros(x*ppm,y*ppm,z*ppm);
    world = build_block(world, 0,0,0,x,y,1,ppm);
    world = build_block(world, x/2-2,y/2-1,0,4,2,5,ppm);


end

function world = build_world_wall2(x,y,z,ppm)
%%This function constructs a world with specified X,Y,Z
dimensions.  It
%%creates a custom environment.  ppm variable represents
the resolution of
```

```
%%the world occupancy grid, for example a ppm value of 5
will create a
%%world occupany array with 5 elements for every meter of
distance.  A
%%higher ppm allows for a finer resolution of object
location but comes
%%with a price of increased memory and computation.

    world = zeros(x*ppm,y*ppm,z*ppm);
    world = build_block(world, 0,0,0,x,y,1,ppm);
    world = build_block(world, 0,y/2-1,0,x,1,8,ppm);


end

function world = build_world_ground(x,y,z,ppm)
%%This function constructs a world with specified X,Y,Z
dimensions.  It
%%creates a custom environment.  ppm variable represents
the resolution of
%%the world occupancy grid, for example a ppm value of 5
will create a
%%world occupany array with 5 elements for every meter of
distance.  A
%%higher ppm allows for a finer resolution of object
location but comes
%%with a price of increased memory and computation.

    world = zeros(x*ppm,y*ppm,z*ppm);
    world = build_block(world, 0,0,0,x,y,1,ppm);
%     world = build_block(world, 0,y/2-1,0,x,2,5,ppm);


end

function world = build_world_hall(x,y,z,ppm)
%%This function constructs a world with specified X,Y,Z
dimensions.  It
%%creates a custom environment.  ppm variable represents
the resolution of
%%the world occupancy grid, for example a ppm value of 5
will create a
%%world occupany array with 5 elements for every meter of
distance.  A
```

```
%%higher ppm allows for a finer resolution of object
location but comes
%%with a price of increased memory and computation.
    hall_width = 1.5;
    world = zeros(x*ppm,y*ppm,z*ppm);
    world = build_block(world, 0,0,0,x,y,1,ppm);
    world = build_block(world, 3,0,0,.2,y,5,ppm);
    world = build_block(world, 8,0,0,.2,y,5,ppm);
    %world = build_block(world,
x/2+hall_width/2,0,0,x/2+hall_width/2+.1,y,1,ppm);

end

function world = build_world_empty(x,y,z,ppm)
%%This function constructs a world with specified X,Y,Z
dimensions.  It
%%creates a custom environment.  ppm variable represents
the resolution of
%%the world occupancy grid, for example a ppm value of 5
will create a
%%world occupany array with 5 elements for every meter of
distance.  A
%%higher ppm allows for a finer resolution of object
location but comes
%%with a price of increased memory and computation.
    world = zeros(x*ppm,y*ppm,z*ppm);


end

function world = build_world_simple2(x,y,z,ppm)
%%This function constructs a world with specified X,Y,Z
dimensions.  It
%%creates a custom environment.  ppm variable represents
the resolution of
%%the world occupancy grid, for example a ppm value of 5
will create a
%%world occupany array with 5 elements for every meter of
distance.  A
%%higher ppm allows for a finer resolution of object
location but comes
%%with a price of increased memory and computation.
    world = zeros(x*ppm,y*ppm,z*ppm);
    world = build_block(world, 0,0,0,x,y,1,ppm);
```

```
    world = build_tree(world, [2/3*x y/2 0], 2,10,ppm);
    world = build_block(world, 0,.6*y,0,x,.5,3,ppm);

end

function world = build_world_urban(x,y,z,ppm)
%%This function constructs a world with specified X,Y,Z
dimensions.  It
%%creates a custom environment.  ppm variable represents
the resolution of
%%the world occupancy grid, for example a ppm value of 5
will create a
%%world occupany array with 5 elements for every meter of
distance.  A
%%higher ppm allows for a finer resolution of object
location but comes
%%with a price of increased memory and computation.
    world = zeros(x*ppm,y*ppm,z*ppm);
    world = build_block(world, 0,0,0,x,y,1,ppm);
    world = build_block(world, 0,0,0,5,5,15,ppm);
    world = build_block(world, 0,6,0,5,6,17,ppm);
    world = build_block(world, 0,12,0,5,6,13,ppm);
    world = build_block(world, 9,0,0,5,6,13,ppm);
    world = build_block(world, 9,0,0,5,8,13,ppm);
    world = build_block(world, 9,12,0,5,5,15,ppm);
    world = build_block(world, 15,2,0,5,3,17,ppm);
    world = build_block(world, 15,12,0,5,3,10,ppm);
end

function worldout = build_block(world,x,y,z,length,
width,height,ppm)
%%This constructor function inserts a block into a world
with specified
%%starting X,Y,Z coordinates and a desired length width and
height, ppm is
%%required to properly fill in the world occupancy array.
    startx = ppm*x + 1;
    starty = ppm*y + 1;
    startz = ppm*z + 1;
    endx = ppm*length + startx;
    endy = ppm*width + starty;
    endz = ppm*height + startz;

    world(startx:endx,starty:endy,startz:endz) = 1;
```

```
    worldout = world;

end

function worldout = build_tree(world,base,r,h,ppm)
%%This function plots a tree at some X,Y coordinate with a
max height
%%desired, the trees branch/leaf radius is r.  ppm is again
required to
%%correctly fill in the world occupancy grid.
    world = pillar(world,.5,h-(2*r),base,ppm);
    center(1) = base(1);
    center(2) = base(2);
    center(3) = h-r;
    world = sphere(world,r,center,ppm);

    worldout = world;
end

function worldout = sphere(world,R,center,ppm)

for phi = -pi:pi/100:pi
    for theta = 0:pi/100:2*pi
        for r=0:.2:R
            x =
round(ppm*(r*cos(theta)*sin(phi)+center(1)));
            y =
round(ppm*(r*sin(theta)*sin(phi)+center(2)));
            z = round(ppm*(r*cos(phi)+center(3)));
            if((x <= 0) || (x <= 0) || (x <= 0))
                keyboard
            end
            world(x,y,z) = 1;
        end
    end
end

worldout = world;
end

function worldout = pillar(world,R,height,pos,ppm)

for h =  1:1/ppm:height
```

```
    for theta = 0:pi/10:2*pi
        for r = 0:.2:R
            x =  round(ppm*(r*cos(theta)+pos(1)));
            y = round(ppm*(r*sin(theta)+pos(2)));
            z = round(ppm*(h+pos(3)));
            world(x,y,z) = 1;
        end
    end
end
worldout = world;
end

function [x, y, z] = plot_world(world, lx, ly, ~,
ppm,pl_tog)
%%This function is used to extract X,Y,Z coordinates of the
occupany array
%%to plot a visual depiction of the real world.  X,Y,Z is
returned so that
%%this computation does not need to be done multiple times.
    area = (lx*ppm+1)*(ly*ppm+1);
    z = (floor((find(world > 0)-1)/area) )/ppm;
    y = (floor(mod((find(world > 0)-
1),area)/(lx*ppm+1)))/ppm;
    x = (mod(mod((find(world > 0)-
1),area),(lx*ppm+1)))/ppm;
    if pl_tog == 1
        plot3(x,y,z,'.');
        hold on
        axis([0 100 0 100 0 100])
    end
end

function [x, y, z] = get_observed_world(world, lx, ly, ~,
ppm,pl_tog)
%%This function is used to extract X,Y,Z coordinates of the
occupany array
%%to plot a visual depiction of the real world.  X,Y,Z is
returned so that
%%this computation does not need to be done multiple times.
    area = (lx*ppm)*(ly*ppm);
    z = ((floor((find(world > .01))/area) ) +1)/ppm ;
    y = ((floor(mod((find(world >
.010)),area)/(lx*ppm)))+1)/ppm;
```

```
    x = ((mod(mod((find(world >
.010)),area),(lx*ppm)))+1)/ppm;
    if pl_tog == 1
        plot3(x,y,z,'.');
        hold on
        axis([0 100 0 100 0 100])
    end
end

function [Xobs,Yobs,Zobs] = plot_obs()
%%This function extracts X,Y,Z coordinates to be used in
plotting of the
%%observation data struct.  The values are returned so that
the
%%computations do not need to be run again.
    global Observations;
    global numObs;

    for i = 1:numObs
        DCM3 = [cos(Observations(i).X(6))
sin(Observations(i).X(6)) 0; -sin(Observations(i).X(6))
cos(Observations(i).X(6)) 0; 0 0 1];
        DCM2 = [cos(Observations(i).X(5)) 0 -
sin(Observations(i).X(5)); 0 1 0; sin(Observations(i).X(5))
0 cos(Observations(i).X(5))];
        DCM1 = [1 0 0; 0 cos(Observations(i).X(4))
sin(Observations(i).X(4)); 0 -sin(Observations(i).X(4))
cos(Observations(i).X(4))];
        bRe = transpose(DCM1*DCM2*DCM3);
        BodyVec = [Observations(i).Obs(1)
Observations(i).Obs(2) Observations(i).Obs(3)]';
        Vec = bRe*BodyVec;
        Xobs(i) = Vec(1) + Observations(i).X(1);
        Yobs(i) = Vec(2) + Observations(i).X(2);
        Zobs(i) = Vec(3) + Observations(i).X(3);
    end

end

function add_source(Pos,activity)
%This function adds a source at the desired position with a
desired
%activity.  It is stored so that counts can be integrated.
    global Sources;
```

```
    global numSources;

    numSources = numSources+1;

    Sources(numSources).Pos = Pos;
    Sources(numSources).Activity = activity;
end

function detect_rad(X,dt)
%This function integrates the counts based off the number
of sources, their
%locations relative to the vehicle location (X), and
integrates the
%activity over some dt.  It saves the counts to be recorded
at a later
%time.
    global Sources;
    global numSources;
    global Counts;
%     global dt;
    global testVar;

    if numSources > 0
        for i = 1:numSources
            dist = norm(X-Sources(i).Pos);
            A = Sources(i).Activity/(dist^2);
            Counts = Counts + get_count_pois(A*dt);
            testVar = testVar + 1;
        end
    end

end

function record_counts(X,t)
%This function stores the count data at and time of the
storing.  The idea
%is that the detector is always counting but the storage of
the count data
%is only done at some specific intervals.  When it is time
to store the
%data this function is called.
    global Counts;
    global RadObs;
    global numRadObs;
```

```
    numRadObs = numRadObs + 1;
    RadObs(numRadObs).Counts = Counts;
    RadObs(numRadObs).Pos = X;
    RadObs(numRadObs).t = t;

    Counts = 0;

end

function count = get_count_pois(lambda)
    if lambda < 16
        k = 0:30;
        P = poissan(lambda,k);
        num = rand(1);
        i = 1;
        done = 0;
        while (done == 0)
            if ( i > length(k))
                done = 1;
                index = 1;
            elseif (num < sum(P(1:i)))

                done = 1;
                index = i;
            end
            i = i +1;
        end

        count = k(index);

    else
        count = round(lambda + normrnd(0,sqrt(lambda)));

    end
end

function P = poissan(lambda,k)
    P = lambda.^(k).*exp(-lambda)./factorial(k);
end

function [Ax Ay Az indices Zmeas] = detect_env3_imu(world,
X, Xk, L, ppm,t,sensX, sensY,
sensZ,Nbeams,Ncubes,cube_info)
```

```
%%This function takes the arrays for the x,y,z coordinates
of the array
%%sensor beams and using them to systematically detect the
local
%%environment.
    global Observations;
    global numObs;
    global accumErr;
    global ang_std;
    Ax = zeros(Nbeams,1);
    Ay = zeros(Nbeams,1);
    Az = zeros(Nbeams,1);
    Zmeas = zeros(Ncubes,1);
    lastnumObs = numObs;

    lx = L(1)*ppm +1;
    ly = L(2)*ppm +1;
    lz = L(3)*ppm +1;
    phi_range = pi/3;
    theta_range = 5*pi/6;
    maxbeam = 5;
    i = 1;
    beam = 0;
    k=1;
    indices(1) = 0;
    DCM3 = [cos(X(6)) sin(X(6)) 0; -sin(X(6)) cos(X(6)) 0;
0 0 1];
    DCM2 = [cos(X(5)) 0 -sin(X(5)); 0 1 0; sin(X(5)) 0
cos(X(5))];
    DCM1 = [1 0 0; 0 cos(X(4)) sin(X(4)); 0 -sin(X(4))
cos(X(4))];
    bRe = transpose(DCM1*DCM2*DCM3);
    for l = 0:10
        for m = 0:10
            beam = beam + 1;
            for n = 1:11
                i = n + m*11 + l*11*11;
                eP = bRe*[sensX(i) sensY(i) sensZ(i)]' +
X(1:3);
                if(~((eP(1) > L(1))  ||(eP(2) > L(2))
||(eP(3) > L(3)) || (eP(1) < 0) || (eP(2) < 0) || (eP(3) <
0)) )

                    ind = round(eP*ppm);
```

```
                    %indx(i) = ind(1);
                    %indy(i) = ind(2);
                    %indz(i) = ind(3);
                    pointer = (ind(3)-1)*lx*ly + (ind(2)-
1)*lx+ind(1);
                    cube_index =
find_measurement_index2_imu(Xk, [sensX(i) sensY(i)
sensZ(i)]');
                    if (world(pointer) == 1)
%                       keyboard;
                       Observations(numObs+1).Xact = X;
                       Observations(numObs+1).X = X; %+
[accumErr' normrnd([0 0 0],[ang_std ang_std ang_std]) 0 0 0
0 0]';
                       Observations(numObs+1).t = t;
                       Observations(numObs+1).Obs =
[sensX(i) sensY(i) sensZ(i)]';
                       Observations(numObs+1).Err =
0;%accumErr;
                       numObs = numObs+1;
                       if (cube_index > 0)
                            Zmeas(cube_index) =
Zmeas(cube_index) + 1;
                            if (Zmeas(cube_index) > 5)
                                 Zmeas(cube_index) = 5;
                            end
                       end
%                        indices(k) = cube_index;
%                      k = k+1;
                       %keyboard;
                       break;
                    elseif (world(pointer) == 0)
                        if (cube_index > 0)
                               Zmeas(cube_index) =
Zmeas(cube_index) - .1;
                        end
                    end
                     indices(k) = cube_index;
                     k = k+1;
               end
          end

       end
   end
```

```
    %keyboard;
    %plot3(x,y,z,'r.');
end

function [NumberOfObservations] =
detect_env3_imu_cleaned(world, X, L, ppm,t,sensX, sensY,
sensZ,Nbeams,Ncubes,Xk)
%%This function takes the arrays for the x,y,z coordinates
of the array
%%sensor beams and using them to systematically detect the
local
%%environment.
    global Observations;
    global numObs;
    global accumErr;
    global ang_std;
    Ox = zeros(Nbeams,1);
    Oy = zeros(Nbeams,1);
    Oz = zeros(Nbeams,1);
    Zmeas = zeros(Ncubes,1);
    NumberOfObservations = numObs;

    lx = L(1)*ppm +1;
    ly = L(2)*ppm +1;
    lz = L(3)*ppm +1;
    phi_range = pi/3;
    theta_range = 5*pi/6;
    maxbeam = 5;
    i = 1;
    beam = 0;
    k=1;
    indices(1) = 0;
    DCM3 = [cos(X(6)) sin(X(6)) 0; -sin(X(6)) cos(X(6)) 0;
0 0 1];
    DCM2 = [cos(X(5)) 0 -sin(X(5)); 0 1 0; sin(X(5)) 0
cos(X(5))];
    DCM1 = [1 0 0; 0 cos(X(4)) sin(X(4)); 0 -sin(X(4))
cos(X(4))];
    bRe = transpose(DCM1*DCM2*DCM3);
    for l = 0:10
        for m = 0:10
            beam = beam + 1;
            for n = 1:11
```

```
                i = n + m*11 + l*11*11;
                eP = bRe*[sensX(i) sensY(i) sensZ(i)]' +
X(1:3);
                if(~((eP(1) > L(1))  ||(eP(2) > L(2))
||(eP(3) > L(3)) || (eP(1) < 0) || (eP(2) < 0) || (eP(3) <
0)) )

                    ind = round(eP*ppm);
                    pointer = (ind(3)-1)*lx*ly + (ind(2)-
1)*lx+ind(1);
                    if (pointer > 0)
%                       keyboard;


                        if (world(pointer) == 1)
                           Observations(numObs+1).Xact = X;
                           Observations(numObs+1).X =
Xk(1:9); %+ [accumErr' normrnd([0 0 0],[ang_std ang_std
ang_std]) 0 0 0 0 0]';
                           Observations(numObs+1).t = t;
                           Observations(numObs+1).Obs =
[sensX(i) sensY(i) sensZ(i)]';
                           Observations(numObs+1).Err =
0;%accumErr;
                           numObs = numObs+1;
                           break;
                        end

                    end

                end
            end

        end
    end
    NumberOfObservations = numObs - NumberOfObservations;
end

function [Pi qi pf ti,which_pi_measured lpi lq
landmarkexists qstr ZObj] = Parse_Observations(Pi, lpi, qi,
lq, qstr, ti, t,
NumberOfObservations,firstPass,landmarkexists)
    %This function parses new observations.  It decides how
many confirmed
```

```
    %landmarks there are based on current and previous
measurements.
    d_determined = 1;
    global Observations;
    global numObs;
    used = 0;
    which_pi_measured = 0;
    pf = [0 0 0]';
    Ztmp = [0 0 0]';

    if (firstPass == 1)
       qi(1:3,1) = [0 0 0]';
       ti = t;
       qstr(1) = 0;
       lq = lq + 1;
    end
    for i  = 1:NumberOfObservations-1
        DCM3 = [cos(Observations(numObs - i).X(6))
sin(Observations(numObs - i).X(6)) 0; -
sin(Observations(numObs - i).X(6)) cos(Observations(numObs
- i).X(6)) 0; 0 0 1];
        DCM2 = [cos(Observations(numObs - i).X(5)) 0 -
sin(Observations(numObs - i).X(5)); 0 1 0;
sin(Observations(numObs - i).X(5)) 0
cos(Observations(numObs - i).X(5))];
        DCM1 = [1 0 0; 0 cos(Observations(numObs - i).X(4))
sin(Observations(numObs - i).X(4)); 0 -
sin(Observations(numObs - i).X(4)) cos(Observations(numObs
- i).X(4))];
        bRe = transpose(DCM1*DCM2*DCM3);
        BodyVec = [Observations(numObs - i).Obs(1)
Observations(numObs - i).Obs(2) Observations(numObs -
i).Obs(3)]';
        Vec = bRe*BodyVec;
        Xobs = Vec(1) + Observations(numObs - i).X(1);
        Yobs = Vec(2) + Observations(numObs - i).X(2);
        Zobs = Vec(3) + Observations(numObs - i).X(3);
        TmpObs = [Xobs Yobs Zobs]';



        if (landmarkexists == 1)
            for k  =  1:lpi
               if  (norm(TmpObs - Pi(:,k)) < d_determined)
```

```
                    used = 1;
                    if (k > length(which_pi_measured))
                        which_pi_measured(k) = 0;
                    end
                    if (which_pi_measured(k) == 0)
                        Ztmp(1:3,k) = Vec;
                        pf(1:3,k) = TmpObs;
                        which_pi_measured(k) = 1;
                    else
                        Ztmp(1:3,k) = (Vec + Ztmp(1:3,k))/2;
                        pf(1:3,k) = (TmpObs + pf(1:3,k))/2;
                        which_pi_measured(k) = 1;
                    end
                    lpf = length(which_pi_measured);
                    break;
                end
            end
            if (used == 0)
                for k  =  1:lq
                    if  (norm(TmpObs - qi(:,k)) <
d_determined)
                        qstr(k) = qstr(k) + 1;
                        qi(1:3,k) = (TmpObs + qi(1:3,k))/2;
                        ti(k) = t;
                        used = 1;

                        break;
                    end

                end
            end
            if (used == 0)
                if (length(qi) < 3)
                    qi(1:3,1) = TmpObs;
                    qstr(1) = + 1;
                    ti(1) = t;
                    lq = lq + 1;
                else
                    qi(1:3,k+1) = TmpObs;
                    qstr(k+1) = + 1;
                    ti(k+1) = t;
                    lq = lq + 1;
                end
```

```
            end

        elseif (landmarkexists ~= 1)
            if (firstPass == 1)
                k  =  2;
                qstr(k) = 1;
                qi(1:3,k) = (TmpObs);
                ti(k) = t;
                used = 1;
                firstPass = 0;
                lq = lq + 1;


            else
                for k  =  1:lq
                        if  (norm(TmpObs - qi(:,k)) <
d_determined)
                            qstr(k) = qstr(k) + 1;
                            qi(1:3,k) = (TmpObs +
qi(1:3,k))/2;
                            ti(k) = t;
                            used = 1;
                            break;
                        end

                end
            end
            firstPass = 0;
            if (used == 0)
                if lq == 0
                   k = 1;
                end
                qi(1:3,k+1) = TmpObs;
                qstr(k+1) = + 1;
                ti(k+1) = t;
                lq = lq + 1;
            end
        end
        fix = 0;
        for k = 1:lq
           if  (qstr(k-fix) > 20)
               if (landmarkexists == 1)
                   Pi(1:3,lpi+1) = qi(:,k-fix);
```

```
                    qi(:,k-fix) = [];
                    ti(k-fix) = [];
                    lpi = lpi +1;
                    lq = lq -1;
                    fix = fix+1;
                else
                     Pi(1:3,1) = qi(:,k-fix);
                     lpi = lpi + 1;
                     landmarkexists = 1;
                     lq = lq -1;
                     fix = fix+1;
                end
            elseif ((t - ti(k-fix)) > 10)
                qi(:,k-fix) = [];
                ti(k-fix) = [];

                lq = lq -1;
                fix = fix+1;
%                 keyboard;
            end
        end
        used = 0;
    end
%     reshape(pf,[3*length(pf),1]);
    ZObj = Ztmp(1:3,find(which_pi_measured == 1));
    if t>27
%         keyboard;
    end
    ZObj = reshape(ZObj, 3*length(ZObj(1,:)), 1);
end

function [sensX sensY sensZ Nbeams] =
build_sensor_array(theta_range, phi_range,maxbeam)
%%This function builds arrays representing the x,y,z
coordinate of a ith
%%beam location.
    Nbeams = 0;
    i = 1;
    for phi = ((pi/2)-
(phi_range/2)):phi_range/10:((pi/2)+(phi_range/2))
        for theta = -
(theta_range/2):theta_range/10:(theta_range/2)
            Nbeams = Nbeams+1;
            for beam = 0:.5:maxbeam
```

```
                sensX(i) = beam*cos(theta)*sin(phi);
                sensY(i) = beam*sin(theta)*sin(phi);
                sensZ(i) = beam*cos(phi);

                i = i +1;
            end

        end
    end
%     plot3(sensX,sensY,sensZ,'r.')
    axis equal
end

function A = cubespace(Wx, Wy, Wz, l)
    xdim = Wx/l;
    ydim = Wy/l;
    zdim = Wz/l;
    A = zeros(xdim,ydim,zdim);
end

function [indice] = find_measurement_index2_imu(X, obs)
    ycubes = 5;
    xcubes = 5;
    j = 1;
    x = (-5:2:5);
    y = (-5:2:5);
    z = (-5:2:5);
    indice(1) = 0;
     DCM3 = [cos(X(6)) sin(X(6)) 0; -sin(X(6)) cos(X(6)) 0;
0 0 1];
     DCM2 = [cos(X(5)) 0 -sin(X(5)); 0 1 0; sin(X(5)) 0
cos(X(5))];
     DCM1 = [1 0 0; 0 cos(X(4)) sin(X(4)); 0 -sin(X(4))
cos(X(4))];
    bRe = transpose(DCM1*DCM2*DCM3);
    for i = 1:length(obs(1,:))

            BodyVec = [obs(1,i) obs(2,i) obs(3,i)]';
             Vec = bRe*BodyVec;
            Xobs = Vec(1);
            Yobs = Vec(2);
            Zobs = Vec(3);
```

```
            if(~((Xobs > x(end))  ||(Yobs > y(end))
||(Zobs > z(end)) || (Xobs < -5) || (Yobs < -5) || (Zobs <
-5)) )
                indx = max(find(Xobs >= x));
                if (isempty(indx))
                   indx = 0;
                end
                indy = max(find(Yobs >= x));
                if (isempty(indy))
                   indy = 0;
                end
                indz = max(find(Zobs >= x));
                if (isempty(indz))
                   indz = 0;
                end

                indice(j) = (indx-1)*ycubes + indy +
xcubes*ycubes*(indz-1);
                j=j+1;
            end
    end
end

function Hk = build_imuHk(Nstates,ObjIndices,
Ncubes,Nmeasurements, RadIndex)

    Hk = zeros(Nmeasurements, Nstates);
    %State position and heading measurements
    Hk(1,1) = 1;
    Hk(2,2) = 1;
    Hk(3,3) = 1;
    Hk(4,4) = 1;
    Hk(5,5) = 1;
    Hk(6,6) = 1;

    %State which cubes are to be measured using the
ObjIndices
    for i = 1:length(ObjIndices)
        if (ObjIndices(i) ~= 0)
            Hk(ObjIndices(i)+6,9+ObjIndices(i)) = 1;
        end
    end

    %State which cube radiation was measured in
```

```
    if (RadIndex ~= 0)
        Hk(end,9+Ncubes+RadIndex) = 1;
    end
end

function Ex =
build_imuEx(Nstates,sigmaVx,sigmaVy,sigmaVz,sigmaYdot,Objer
r,Raderr,deltaT)
    %handle spatial sigmas for the vehicle state portion
    sigmaX = deltaT*sigmaVx;
    sigmaY = deltaT*sigmaVx;
    sigmaZ = deltaT*sigmaVx;
    sigmaRoll = .1;
    sigmaPitch = .1;
    sigmaYaw = deltaT*sigmaYdot;
    Ex = zeros(Nstates,Nstates);

    Ex(1,1) = sigmaX*sigmaX;
    Ex(2,2) = sigmaY*sigmaY;
    Ex(3,3) = sigmaZ*sigmaZ;
    Ex(4,4) = sigmaRoll*sigmaRoll;
    Ex(5,5) = sigmaPitch*sigmaPitch;
    Ex(6,6) = sigmaYaw*sigmaYaw;
    Ex(7,7) = sigmaVx*sigmaVx;
    Ex(8,8) = sigmaVy*sigmaVy;
    Ex(9,9) = sigmaVz*sigmaVz;


    for i = 10:(10+(Nstates-10)/2)
       Ex(i,i) = Objerr^2;
       Ex(i+(Nstates-9)/2,i+(Nstates-9)/2) = Raderr^2;
    end

end

function Ez = build_imuEz(Nmeasurements,Nbeams
,imuErr,ObjMeasErr, RadMeasErr)
    Ez = zeros(Nmeasurements,Nmeasurements);
    Ez(1,1) = imuErr(1)^2;
    Ez(2,2) = imuErr(1)^2;
    Ez(3,3) = imuErr(1)^2;
    Ez(4,4) = imuErr(2)^2;
    Ez(5,5) = imuErr(2)^2;
    Ez(6,6) = imuErr(2)^2;
```

```
    for i = 7:Nmeasurements-1
        Ez(i,i) = ObjMeasErr^2;
    end
    Ez(end,end) = RadMeasErr^2;

end

function ZMeas =
handle_Measurements_imu(X,Nmeasurements,Err,tmpMeas,counts)
    ZMeas = zeros(Nmeasurements,1);
    ZMeas(1:6) = [X(1)+normrnd(0,Err(1))
X(2)+normrnd(0,Err(2)) X(3)+normrnd(0,Err(3))
X(4)+normrnd(0,Err(4)) X(5)+normrnd(0,Err(5))
X(6)+normrnd(0,Err(6))]';
    ZMeas(7:end-1) = tmpMeas;
    ZMeas(end) = counts;
end

function [is_valid_o Xo Yo Zo STR is_valid_r Xr Yr Zr
COUNTS] =
get_xyz_from_state_imu(state_vec,xcubes,ycubes,zcubes,dpc,N
cubes)
    %Find the Objects that are present
    ind = find(state_vec(10:9+Ncubes) ~= 0);
    z = ceil(ind/(xcubes*ycubes));
    x = ceil((ind - ((z-1)*xcubes*ycubes))/ycubes);
    y = ind - ((z-1)*xcubes*ycubes) - (x-1)*ycubes;
    z = z*dpc;
    y = y*dpc;
    x = x*dpc;
    str = state_vec(9+ind);
    indnew = find(str>0);
    if (isempty(indnew))
        is_valid_o = 0;
        Xo = 0;
        Yo = 0;
        Zo = 0;
        STR = 0;
    else
        is_valid_o = 1;
        Xo = x(indnew);
        Yo = y(indnew);
        Zo = z(indnew);
        STR = 100*str(indnew);
```

```
        if ~isempty(find(STR > 100))
            STR(find(STR > 100)) = 100;
        end
    end

    %Find the Radiation Objects that are present
    ind = find(state_vec(9+Ncubes+1:end) ~= 0);
    z = ceil(ind/(xcubes*ycubes));
    x = ceil((ind - ((z-1)*xcubes*ycubes))/ycubes);
    y = ind - ((z-1)*xcubes*ycubes) - (x-1)*ycubes;
    z = z*dpc;
    y = y*dpc;
    x = x*dpc;
    counts = state_vec(9+Ncubes+ind);
    indnew = find(counts>0);
    if (isempty(indnew))
        is_valid_r = 0;
        Xr = 0;
        Yr = 0;
        Zr = 0;
        COUNTS = 0;
    else
        is_valid_r = 1;
        Xr = x(indnew);
        Yr = y(indnew);
        Zr = z(indnew);
        COUNTS = counts(indnew);
    end
end

function [will_hit x] = check_traj(X,xdes,Vdir, Xobs, Yobs,
Zobs)
    x = X(1:3);
    will_hit = 0;
    while (norm(x-xdes) >= .75)
        if ( sum((ceil(x(1)) == Xobs) & (ceil(x(2)) ==
Yobs) & (ceil(x(3)) == Zobs)) == 1)
            will_hit = 1;
            break;
        else
            will_hit = 0;
            x = x +Vdir*.5;
        end
        if (norm(x) > 30)
```

```
            break;
        end

    end
end

function [will_hit x] = check_traj2(X,xdes, Vdir, Pi, lpi)
    %%This function checks to see if the trajectory comes
to within 1m of
    %%any identified landmark in memory.
    x = X(1:3);
    will_hit = 0;
%     keyboard;
    while (norm(x-xdes) >= .75)
        for i  =  1: lpi
          if (norm(x-Pi(:,i)) < 1)
              will_hit = 1;
              break;
          end
        end
        x = x + Vdir*.5;
        if (norm(x) > 30)
           break;
        elseif (will_hit == 1)
           break;
        end


    end
end

function new_way = get_new_way(xCol,X,xdes,Vdir, Xobs,
Yobs, Zobs)
    rx = abs(xCol(1) - (Xobs-.5));
    ry = abs(xCol(2) - (Yobs-.5));
    rz = abs(xCol(3) - (Zobs-.5));
    rx1 = abs(xCol(1)-.1 - (Xobs-.5));
    ry1 = abs(xCol(2)-.1 - (Yobs-.5));
    rz1 = abs(xCol(3)-.1 - (Zobs-.5));
    rx2 = abs(xCol(1)+.1 - (Xobs-.5));
    ry2 = abs(xCol(2)+.1 - (Yobs-.5));
    rz2 = abs(xCol(3)+.1 - (Zobs-.5));

    tmp1 = [rx1 ry rz];
```

```
    tmp2 = [rx2 ry rz];
    sum1 = 0;
    sum2 = 0;
    tmp3 = [rx ry1 rz];
    tmp4 = [rx ry2 rz];
    sum3 = 0;
    sum4 = 0;
    tmp5 = [rx ry rz1];
    tmp6 = [rx ry rz2];
    sum5 = 0;
    sum6 = 0;
    for i  = 1:length(tmp1(:,1))
        sum1 = sum1 + 1/norm(tmp1(i,:));
        sum2 = sum2 + 1/norm(tmp2(i,:));
        sum3 = sum3 + 1/norm(tmp3(i,:));
        sum4 = sum4 + 1/norm(tmp4(i,:));
        sum5 = sum5 + 1/norm(tmp5(i,:));
        sum6 = sum6 + 1/norm(tmp6(i,:));
    end
    grad(1) = (sum2 - sum1)/.2;
    grad(2) = (sum4 - sum3)/.2;
    grad(3) = (sum6 - sum5)/.2;

    new_way = xCol + (-grad'/norm(grad)*3);

%     keyboard;
end

function new_way = get_new_way2(xCol,X,xdes,Vdir, Pi, lpi)
    rx = abs(xCol(1) - (Pi(1,:)-.5));
    ry = abs(xCol(2) - (Pi(2,:)-.5));
    rz = abs(xCol(3) - (Pi(3,:)-.5));
    rx1 = abs(xCol(1)-.1 - (Pi(1,:)-.5));
    ry1 = abs(xCol(2)-.1 - (Pi(2,:)-.5));
    rz1 = abs(xCol(3)-.1 - (Pi(3,:)-.5));
    rx2 = abs(xCol(1)+.1 - (Pi(1,:)-.5));
    ry2 = abs(xCol(2)+.1 - (Pi(2,:)-.5));
    rz2 = abs(xCol(3)+.1 - (Pi(3,:)-.5));

    tmp1 = [rx1 ry rz];
    tmp2 = [rx2 ry rz];
    sum1 = 0;
    sum2 = 0;
    tmp3 = [rx ry1 rz];
```

```
    tmp4 = [rx ry2 rz];
    sum3 = 0;
    sum4 = 0;
    tmp5 = [rx ry rz1];
    tmp6 = [rx ry rz2];
    sum5 = 0;
    sum6 = 0;
    for i  = 1:length(tmp1(:,1))
        sum1 = sum1 + 1/norm(tmp1(i,:));
        sum2 = sum2 + 1/norm(tmp2(i,:));
        sum3 = sum3 + 1/norm(tmp3(i,:));
        sum4 = sum4 + 1/norm(tmp4(i,:));
        sum5 = sum5 + 1/norm(tmp5(i,:));
        sum6 = sum6 + 1/norm(tmp6(i,:));
    end
    grad(1) = (sum2 - sum1)/.2;
    grad(2) = (sum4 - sum3)/.2;
    grad(3) = (sum6 - sum5)/.2;

    new_way = xCol + (-grad'/norm(grad)*3);

%       keyboard;
end

function X = f1(Xk,Uk,deltaT,Nstates,Ncubes)
    X = zeros(Nstates,1);

    %Compute the changes in position, velocity, and
orientation
    deltaX = Xk(7)*deltaT + .5*Uk(1)*deltaT*deltaT;
    deltaY = Xk(8)*deltaT + .5*Uk(2)*deltaT*deltaT;
    deltaZ = Xk(9)*deltaT + .5*Uk(3)*deltaT*deltaT;
    deltaRoll = Uk(4)*deltaT;
    deltaPitch = Uk(5)*deltaT;
    deltaYaw = Uk(6)*deltaT;
    deltaVx = Uk(1)*deltaT;
    deltaVy = Uk(2)*deltaT;
    deltaVz = Uk(3)*deltaT;

    %X,Y,Z translation estimate
    X(1) = Xk(1) + deltaX;
    X(2) = Xk(2) + deltaY;
    X(3) = Xk(3) + deltaZ;
```

```
    %Roll,Pitch,Yaw rotation estimates
    X(4) = Xk(4) + deltaRoll;
    X(5) = Xk(5) + deltaPitch;
    X(6) = Xk(6) + deltaYaw;

    %Vx,Vy,Vz estimates
    X(7) = Xk(7) + deltaVx;
    X(8) = Xk(8) + deltaVy;
    X(9) = Xk(9) + deltaVz;

    if deltaX >= 0
        Xchange = -1;
    else
        Xchange = 1;
    end
    if deltaY >= 0
        Ychange = -1;
    else
        Ychange = 1;
    end
    if deltaZ >= 0
        Zchange = -1;
    else
        Zchange = 1;
    end

    objWorld = reshape(Xk(10:10+Ncubes-1),5,5,5);
    radWorld = reshape(Xk(10+Ncubes:end),5,5,5);
    newObjWorld = zeros(5,5,5);
    newRadWorld = zeros(5,5,5);

    for z  = 1:5
        for y = 1:5
           for x = 1:5
               if(((x == 1) && (Xchange == -1)) || ((x==5)
&& (Xchange == 1)))
                   %Get value from observed world
                   Bxo = 0;
                   Bxr = 0;
               else
                   Bxo = objWorld(x+Xchange,y,z);
                   Bxr = radWorld(x+Xchange,y,z);
               end
```

```
                if(((y == 1) && (Ychange == -1)) || ((y==5)
&& (Ychange == 1)))
                    %Get value from observed world
                    Byo = 0;
                    Byr = 0;
                else
                    Byo = objWorld(x,y+Ychange,z);
                    Byr = radWorld(x,y+Ychange,z);
                end
                if(((z == 1) && (Zchange == -1)) || ((z==5)
&& (Zchange == 1)))
                    %Get value from observed world
                    Bzo = 0;
                    Bzr = 0;
                else
                    Bzo = objWorld(x,y,z+Zchange);
                    Bzr = radWorld(x,y,z+Zchange);
                end

                if(deltaX > 2)
                    disp('deltaX > 2m, address issue');
                end
                if(deltaY > 2)
                    disp('deltaY > 2m, address issue');
                end
                if(deltaZ > 2)
                    disp('deltaZ > 2m, address issue');
                end

                newObjWorld(x,y,z) = (norm(deltaX)*Bxo +
norm(deltaY)*Byo + norm(deltaZ)*Bzo + objWorld(x,y,z)*(6 -
norm(deltaX) - norm(deltaY) - norm(deltaZ)))/6;
                newRadWorld(x,y,z) = (norm(deltaX)*Bxr +
norm(deltaY)*Byr + norm(deltaZ)*Bzr + radWorld(x,y,z)*(6 -
norm(deltaX) - norm(deltaY )- norm(deltaZ)))/6;
            end
        end
    end
    beforeSum = sum(objWorld(:));
    afterSum = sum(newObjWorld(:));
    X(10:10+Ncubes-1) = reshape(newObjWorld,125,1);
    X(10+Ncubes:end) = reshape(newRadWorld,125,1);
end
```

```
function X = f2(t,f)
    deltaT = .1;
    Nstates = 259;
    Ncubes = 125;
    X = zeros(Nstates,1);
    Xk = f(1:259);
    Uk = f(260:265);
    %Compute the changes in position, velocity, and
orientation
    deltaX = Xk(7)*deltaT + .5*Uk(1)*deltaT*deltaT;
    deltaY = Xk(8)*deltaT + .5*Uk(2)*deltaT*deltaT;
    deltaZ = Xk(9)*deltaT + .5*Uk(3)*deltaT*deltaT;
    deltaRoll = Uk(4)*deltaT;
    deltaPitch = Uk(5)*deltaT;
    deltaYaw = Uk(6)*deltaT;
    deltaVx = Uk(1)*deltaT;
    deltaVy = Uk(2)*deltaT;
    deltaVz = Uk(3)*deltaT;

    %X,Y,Z translation estimate
    X(1) = Xk(1) + deltaX;
    X(2) = Xk(2) + deltaY;
    X(3) = Xk(3) + deltaZ;

    %Roll,Pitch,Yaw rotation estimates
    X(4) = Xk(4) + deltaRoll;
    X(5) = Xk(5) + deltaPitch;
    X(6) = Xk(6) + deltaYaw;

    %Vx,Vy,Vz estimates
    X(7) = Xk(7) + deltaVx;
    X(8) = Xk(8) + deltaVy;
    X(9) = Xk(9) + deltaVz;

    if deltaX >= 0
        Xchange = -1;
    else
        Xchange = 1;
    end
    if deltaY >= 0
        Ychange = -1;
    else
        Ychange = 1;
    end
```

```
    if deltaZ >= 0
        Zchange = -1;
    else
        Zchange = 1;
    end

    objWorld = reshape(Xk(10:10+Ncubes-1),5,5,5);
    radWorld = reshape(Xk(10+Ncubes:end),5,5,5);
    newObjWorld = zeros(5,5,5);
    newRadWorld = zeros(5,5,5);

    for z  = 1:5
        for y = 1:5
           for x = 1:5
               if(((x == 1) && (Xchange == -1)) || ((x==5)
&& (Xchange == 1)))
                   %Get value from observed world
                   Bxo = 0;
                   Bxr = 0;
               else
                   Bxo = objWorld(x+Xchange,y,z);
                   Bxr = radWorld(x+Xchange,y,z);
               end
               if(((y == 1) && (Ychange == -1)) || ((y==5)
&& (Ychange == 1)))
                   %Get value from observed world
                   Byo = 0;
                   Byr = 0;
               else
                   Byo = objWorld(x,y+Ychange,z);
                   Byr = radWorld(x,y+Ychange,z);
               end
               if(((z == 1) && (Zchange == -1)) || ((z==5)
&& (Zchange == 1)))
                   %Get value from observed world
                   Bzo = 0;
                   Bzr = 0;
               else
                   Bzo = objWorld(x,y,z+Zchange);
                   Bzr = radWorld(x,y,z+Zchange);
               end

               if(deltaX > 2)
                   disp('deltaX > 2m, address issue');
```

```
                end
                if(deltaY > 2)
                    disp('deltaY > 2m, address issue');
                end
                if(deltaZ > 2)
                    disp('deltaZ > 2m, address issue');
                end

                newObjWorld(x,y,z) = (norm(deltaX)*Bxo +
norm(deltaY)*Byo + norm(deltaZ)*Bzo + objWorld(x,y,z)*(3 -
norm(deltaX) - norm(deltaY) - norm(deltaZ)))/6;
                newRadWorld(x,y,z) = (norm(deltaX)*Bxr +
norm(deltaY)*Byr + norm(deltaZ)*Bzr + radWorld(x,y,z)*(3 -
norm(deltaX) - norm(deltaY) - norm(deltaZ)))/6;
            end
        end
    end

    X(10:10+Ncubes-1) = reshape(newObjWorld,125,1);
    X(10+Ncubes:end) = reshape(newRadWorld,125,1);
end

function [Ak, Bk] = get_Ak_jacobian(X,U)
    tmpVec = [X' U']';
    X0 = f2(0,tmpVec);
    thresh = 1e-6;
    [dFdX,fac] = numjac(@f2,1,tmpVec,X0,thresh,[],0);

    Ak = dFdX(:,1:259);
    Bk = dFdX(:,260:end);

end

function world_out =
save_KF_world(X,observedWorld,world_dim,ppm)
     lx = world_dim(1);
     ly = world_dim(2);
     lz = world_dim(3);

     x = ceil(X(1)*ppm);
     y = ceil(X(2)*ppm);
     z = ceil(X(3)*ppm)+3;
```

```
     observedWorld(x-2:x+2,y-2:y+2,z-2:z+2) =
reshape(X(10:10+125-1),5,5,5);

     world_out = observedWorld;
end

function Xknew = fix_state_vector(Xk, NL, Pi)
    if (length(Xk) < (9+3+(3*NL)))
        newLandmarkCount = NL - (length(Xk) - 13)/3;
        Xknew = Xk(1:length(Xk) - 3);
        for i  =  NL + 1 - newLandmarkCount:NL
            Xknew(9+3*i-2) = Pi(1,i);      %X-coord of the
new Landmark
            Xknew(9+3*i-1) = Pi(2,i);      %Y-coord of the
new Landmark
            Xknew(9+3*i) = Pi(3,i);        %Z-coord of the
new Landmark

        end
        Xknew(9+3*NL+1:9+3*NL+4) = Xk(length(Xk) - 3:end);

    else
        Xknew = Xk;
    end
end

function X = TransitionModel(Xk,Uk,deltaT)
    X = zeros(length(Xk),1);

    %Compute the changes in position, velocity, and
orientation
    deltaX = Xk(7)*deltaT + .5*Uk(1)*deltaT*deltaT;
    deltaY = Xk(8)*deltaT + .5*Uk(2)*deltaT*deltaT;
    deltaZ = Xk(9)*deltaT + .5*Uk(3)*deltaT*deltaT;
    deltaRoll = Uk(4)*deltaT;
    deltaPitch = Uk(5)*deltaT;
    deltaYaw = Uk(6)*deltaT;
    deltaVx = Uk(1)*deltaT;
    deltaVy = Uk(2)*deltaT;
    deltaVz = Uk(3)*deltaT;

    %X,Y,Z translation estimate
    X(1) = Xk(1) + deltaX;
    X(2) = Xk(2) + deltaY;
```

```
    X(3) = Xk(3) + deltaZ;

    %Roll,Pitch,Yaw rotation estimates
    X(4) = Xk(4) + deltaRoll;
    X(5) = Xk(5) + deltaPitch;
    X(6) = Xk(6) + deltaYaw;

    %Vx,Vy,Vz estimates
    X(7) = Xk(7) + deltaVx;
    X(8) = Xk(8) + deltaVy;
    X(9) = Xk(9) + deltaVz;

    X(10:end) = Xk(10:end);

end

function X = TransitionModel2(t,f)

    deltaT = .1;
    Xk = f(1:length(f) - 6);
    Uk = f(length(f) - 5:length(f));


    %Compute the changes in position, velocity, and
orientation
    deltaX = Xk(7)*deltaT + .5*Uk(1)*deltaT*deltaT;
    deltaY = Xk(8)*deltaT + .5*Uk(2)*deltaT*deltaT;
    deltaZ = Xk(9)*deltaT + .5*Uk(3)*deltaT*deltaT;
    deltaRoll = Uk(4)*deltaT;
    deltaPitch = Uk(5)*deltaT;
    deltaYaw = Uk(6)*deltaT;
    deltaVx = Uk(1)*deltaT;
    deltaVy = Uk(2)*deltaT;
    deltaVz = Uk(3)*deltaT;

    %X,Y,Z translation estimate
    X(1) = Xk(1) + deltaX;
    X(2) = Xk(2) + deltaY;
    X(3) = Xk(3) + deltaZ;

    %Roll,Pitch,Yaw rotation estimates
    X(4) = Xk(4) + deltaRoll;
    X(5) = Xk(5) + deltaPitch;
    X(6) = Xk(6) + deltaYaw;
```

```
    %Vx,Vy,Vz estimates
    X(7) = Xk(7) + deltaVx;
    X(8) = Xk(8) + deltaVy;
    X(9) = Xk(9) + deltaVz;

    X(10:length(Xk)) = Xk(10:end);

end

function [Ak, Bk] = get_Ak_jacobian2(X,U)
    tmpVec = [X' U']';
    X0 = TransitionModel2(0,tmpVec);
    thresh = 1e-6;
    [dFdX,fac] =
numjac(@TransitionModel2,1,tmpVec,X0,thresh,[],0);

    Ak = dFdX(:,1:length(X));
    Bk = dFdX(:,length(X)+1:end);

end


function [Z,RadErr]  = ObservationModel(pf,
which_pi_measured, Xk)
    global gotRadMeasurements;

    lpf = length(pf);

    %Estimate state position and heading measurements
    Z(1:6) = Xk(1:6);
    lengthXk = length(Xk);
%     keyboard;
    ind = find(which_pi_measured==1);

%     DCM3 = [cos(X(6)) sin(X(6)) 0; -sin(X(6)) cos(X(6))
0; 0 0 1];
%     DCM2 = [cos(X(5)) 0 -sin(X(5)); 0 1 0; sin(X(5)) 0
cos(X(5))];
%     DCM1 = [1 0 0; 0 cos(X(4)) sin(X(4)); 0 -sin(X(4))
cos(X(4))];
%     bRe = DCM1*DCM2*DCM3;


    for i  =  1: length(ind)
```

```
%         Z(6+3*i-2:6+3*i) =  bRe*(X(9+(ind(i)-1)*3 +
1:9+ind(i)*3) - X(1:3));     %earth frame vectors landmark
- vehicle position rotated to body frame
            Z(6+3*i-2:6+3*i) = (Xk(9+(ind(i)-1)*3 +
1:9+ind(i)*3) - Xk(1:3));
    end
    rSquared = (Xk(lengthXk-3) - Xk(1))^2 + (Xk(lengthXk-2)
- Xk(2))^2 + (Xk(lengthXk-1) - Xk(3))^2;
    RadMeasure = Xk(end)/rSquared;
    if (gotRadMeasurements == 1)

        Z(end+1) = RadMeasure;
    else
        Z(end+1) = 0;
    end
    Z = Z';
    RadErr = sqrt(RadMeasure);
end

function Z = ObservationModel2(t, Xk)
    global pf;
    global which_pi_measured;
    global gotRadMeasurements;

    %Estimate state position and heading measurements
    Z(1:6) = Xk(1:6);
    lengthXk = length(Xk);
%     keyboard;
    ind = find(which_pi_measured==1);

%     DCM3 = [cos(X(6)) sin(X(6)) 0; -sin(X(6)) cos(X(6))
0; 0 0 1];
%     DCM2 = [cos(X(5)) 0 -sin(X(5)); 0 1 0; sin(X(5)) 0
cos(X(5))];
%     DCM1 = [1 0 0; 0 cos(X(4)) sin(X(4)); 0 -sin(X(4))
cos(X(4))];
%     bRe = DCM1*DCM2*DCM3;


    for i  =  1: length(ind)
%         Z(6+3*i-2:6+3*i) =  bRe*(X(9+(ind(i)-1)*3 +
1:9+ind(i)*3) - X(1:3));     %earth frame vectors landmark
- vehicle position rotated to body frame
```

```
    Z(6+3*i-2:6+3*i) = (Xk(9+(ind(i)-1)*3 + 1:9+ind(i)*3) -
Xk(1:3));
    end
    if (gotRadMeasurements == 1)
        rSquared = (Xk(lengthXk-3) - Xk(1))^2 +
(Xk(lengthXk-2) - Xk(2))^2 + (Xk(lengthXk-1) - Xk(3))^2;
        RadMeasure = Xk(end)/rSquared;
        Z(end+1) = RadMeasure;
    else
        Z(end+1) = 0;
    end
    Z = Z';
end

function H = get_Hk_jacobian(X,pf1,which_pi_measured1)
    global pf;
    global which_pi_measured;
    pf = pf1;
    which_pi_measured = which_pi_measured1;
    X0 = ObservationModel2(0,X);
    thresh = 1e-8;
    [H,fac] =
numjac(@ObservationModel2,1,X,X0,thresh,[],0);

%     keyboard;
end

function [X, Obj, Rad] = break_state_vector(Xk)
    X = Xk(1:9);
    tmpVec = Xk(10:length(Xk)-4);
    NL =  length(tmpVec)/3;
    Obj =  zeros(3,NL);
    for i  = 1:NL
        Obj(:,i) = Xk(9+(i-1)*3+1:9+(i)*3);
    end
    Rad = Xk(length(Xk)-3:end);

end

function Ez = build_Ez(NumMeasurements, GPSerr, CompassErr,
ObjErr, RadErr)
    Ez =  zeros (NumMeasurements,NumMeasurements);
    Ez(1,1) = GPSerr^2;
    Ez(2,2) = GPSerr^2;
```

```
    Ez(3,3) = GPSerr^2;
    Ez(4,4) = CompassErr^2;
    Ez(5,5) = CompassErr^2;
    Ez(6,6) = CompassErr^2;
    for i = 7:NumMeasurements-1
        Ez(i,i) = ObjErr^2;
    end
    Ez(end,end) = RadErr^2;
end

function Ex = build_Ex(NL, XYZErr,VXYZerr, VRPYerr, ObjErr,
RadErr, deltaT)
    sigmaX = deltaT*VXYZerr;
    sigmaY = deltaT*VXYZerr;
    sigmaZ = deltaT*VXYZerr;
    sigmaRoll = .1;
    sigmaPitch = .1;
    sigmaYaw = deltaT*VRPYerr;
    Ex = zeros(NL*3+12,NL+12);

    Ex(1,1) = XYZErr;
    Ex(2,2) = XYZErr;
    Ex(3,3) = XYZErr;
    Ex(4,4) = sigmaRoll*sigmaRoll;
    Ex(5,5) = sigmaPitch*sigmaPitch;
    Ex(6,6) = sigmaYaw*sigmaYaw;
    Ex(7,7) = VXYZerr*VXYZerr;
    Ex(8,8) = VXYZerr*VXYZerr;
    Ex(9,9) = VXYZerr*VXYZerr;


    for i = 10:(9+3*NL)
       Ex(i,i) = ObjErr^2;
    end
    Ex(10+3*NL,10+3*NL) = RadErr(1)^2;
    Ex(11+3*NL,11+3*NL) = RadErr(1)^2;
    Ex(12+3*NL,12+3*NL) = RadErr(1)^2;
    Ex(13+3*NL,13+3*NL) = RadErr(2)^2;
end

function Pout = Fix_Pk(P,Xk)
    if (length(P(:,1)) ~= length(Xk))
        LengthOfP = length(P(:,1));
        LengthOfXk = length(Xk);
```

```
        diff = LengthOfXk - LengthOfP;
        tmpMat1 = P(1:LengthOfP-4,1:LengthOfP-4);
        tmpMat2 = P(1:LengthOfP-4,LengthOfP-3:LengthOfP);
        tmpMat3 = P(LengthOfP-3:LengthOfP,1:LengthOfP-4);
        tmpMat4 = P(LengthOfP-3:LengthOfP,LengthOfP-
3:LengthOfP);
        Pout = zeros(length(Xk));
        Pout(1:LengthOfP-4,1:LengthOfP-4) = tmpMat1;
        Pout(1:LengthOfP-4,LengthOfP-3+diff:LengthOfP+diff)
= tmpMat2;
        Pout(LengthOfP-3+diff:LengthOfP+diff,1:LengthOfP-4)
= tmpMat3;
        Pout(LengthOfXk-3:LengthOfXk,LengthOfXk-
3:LengthOfXk) = tmpMat4;
    else
        Pout = P;
    end

end

function Pout = build_Po(Xk)
    Pout = zeros(length(Xk));
     Pout(length(Xk),length(Xk)) = 100;
     Pout(length(Xk)-1,length(Xk)-1) = 20;
     Pout(length(Xk)-2,length(Xk)-2) = 20;
     Pout(length(Xk)-3,length(Xk)-3) = 20;
end

function Z = GetMeasurement(X, Err, Zobj,
measurementLength,GotRadMeasurement)
    global RadObs;
    global numRadObs;

    Z(1) = X(1) + normrnd(0,Err(1));
    Z(2) = X(2) + normrnd(0,Err(1));
    Z(3) = X(3) + normrnd(0,Err(1));
    Z(4) = X(4) + normrnd(0,Err(2));
    Z(5) = X(5) + normrnd(0,Err(2));
    Z(6) = X(6) + normrnd(0,Err(2));
    Z(7:measurementLength-1) = Zobj;
    if (GotRadMeasurement == 1)
        Z(measurementLength) = RadObs(numRadObs).Counts;
    else
        Z(measurementLength) = 0;
```

```
    end
    Z = Z';
end
```

APPENDIX B: RK4 CODE

```
%This function implements the 4th-order Runge Kutta method
for the system of ODEs
%dv/dt = f(t,v,parms) from time t=a until time t=b,
%using a step size of h.  The initial condition is v(0) =
v0.
%The results are returned in a matrix, where the column 1
holds the time values,
%and then the k'th column holds the (k-1)'st state variable
over time
%I took the .R code from your site and rewrote for MATLAB
function A = RK4(eqn, xo, a, h, numSteps, param)

numCols = length(xo) + 1;
nrows = numSteps + 1;
ncol = numCols;
nrows = int16(nrows);
ncol = int16(ncol);
A = zeros(nrows,ncol);
A(1,1) = a;
A(1,2:end) = xo(:);
if (length(param) == 1) && (param == 0)
    for i = 1:numSteps
        A(i+1,1) = a+i*h;
        s1 = eqn(A(i,1), A(i,2:numCols))';
        s2 = eqn(A(i,1) + h/2, A(i,2:numCols) + h/2*s1)';
        s3 = eqn(A(i,1) + h/2, A(i,2:numCols) + h/2*s2)';
        s4 = eqn(A(i,1) + h, A(i,2:numCols) + h*s3)';
        A(i+1,2:numCols) = A(i,2:numCols) +h/6*(s1 + 2*s2 +
2*s3 + s4);
    end
else
    for i = 1:numSteps
        A(i+1,1) = a+i*h;
        s1 = eqn(A(i,1), A(i,2:numCols), param)';
        s2 = eqn(A(i,1) + h/2, A(i,2:numCols) + h/2*s1,
param)';
        s3 = eqn(A(i,1) + h/2, A(i,2:numCols) + h/2*s2,
param)';
        s4 = eqn(A(i,1) + h, A(i,2:numCols) + h*s3,
param)';
        A(i+1,2:numCols) = A(i,2:numCols) +h/6*(s1 + 2*s2 +
2*s3 + s4);
```

```
    end
end
```

BIOGRAPHY OF THE AUTHOR

Shawn Brackett was born December 12, 1991 in Lewiston Maine. He was raised by his two parents, Linda and Richard Brackett, and had one sister, Andrea Brackett. He attended private school until the 4th grade, then transferred into the public school system, eventually graduating from Edward Little High School. During his time in high school he played lacrosse, his first year he was the team captain for the J.V. team, and eventually would become a captain for the varsity team by his senior year. Not knowing what to do after high school, he applied to the University of Maine, and was accepted as an undecided Engineering student. Eventually he would make the decision to pursue a degree in Engineering Physics. His senior year he met his now wife Irene Brackett, and decided to attend the University of Maine for his graduate degree in Engineering Physics. Shawn is a candidate for the Master of Engineering degree in Engineering Physics from the University of Maine in August 2017.